COLLINS GEM

MICRO FACTS

commodore

C-64

An A-Z of essential facts

COLLINS
GEM

MICRO FACTS
COMMODORE
64

COLLINS
GEM

MICRO FACTS

COMMODORE
64

Simon Beesley

Collins
London and Glasgow

Foreword

This 'Micro Facts' is a comprehensive reference guide to the Commodore 64. It is organised on the principle of a dictionary and with over 300 entries covers almost every aspect of the computer.

The intention of this book is that it should be both practical and easy to use. Although the Commodore 64 is one of the most powerful home computers around, many of its facilities are not readily available. Commodore BASIC – unlike other versions of the language – has no commands to handle sound and graphics. To display *sprites*, for example, the user needs to *POKE* values into a series of memory locations.

All these features are explained in detail. The memory locations which control them are presented in tables alongside the relevant entries. The alphabetical order ensures that the reader will have no problem in finding any particular reference.

The basic rule for finding information in this book is to look under the most obvious word. If that fails – and it may only be 'obvious' to you – then try a related word. Where a topic has been developed further under another

heading, **bold print** in the text indicates the heading of a separate entry. Related topics are also extensively cross-referenced in this way.

All the BASIC keywords have been included, and are usually accompanied by a program example. In addition, the book covers the machine code instructions for the computer's 6510 microprocessor – the same instructions as those used with the 6502 microprocessor in some other home computers. Each instruction is given its own entry, together with a table showing the different forms it can take.

As far as possible, jargon has been avoided, but where specialized terms are needed, they are used, and explained elsewhere under their own entries.

'Commodore' and 'VIC' are registered trade marks of Commodore Business Machines. The names or terms 'Atari', 'Centronics', 'CP/M', 'Prestel' and 'Z80' are also legally protected and exclusive to their respective owners.

abbreviations Most of the BASIC keywords can be entered as abbreviations. Usually these take the form of a letter followed by a **graphics character**. The most commonly used abbreviation is the question mark to replace the PRINT keyword. For example '? PEEK (197)' is the same as 'PRINT PEEK(197)'. Each keyword is stored in memory as a 1-byte **token**, so entering an abbreviation does not save space in memory, but it makes it possible to put more than 80 characters on a program line.

When a program is LISTed abbreviations are expanded to their normal keywords.

ABS (ABSolute value) A numeric function which turns negative numbers into positive numbers leaving positive numbers unchanged. Thus the absolute value of -3.75 is 3.75 while the absolute value of 8.3 is 8.3. It requires its argument to be placed in parentheses, as in '10 Y=ABS(X)' or '10 PRINT ABS(5×N)'. One of the uses of the ABS function is for calculating the difference between two numbers when you do not know which is larger. If, for example, 'M=5' and 'N=9', then 'M−N' equals '−4', but 'ABS(M−N)' returns a posi-

tive value.

Associated keyword: **SGN**.

absolute addressing Treats the two bytes following the **op code** as the address of a byte in memory, e.g. 'STA $0423' stores the value of the accumulator at location $0423.

accumulator The most frequently used register in the 6510 microprocessor. All arithmetic and logical operations are carried out in the accumulator.

ADC The only 6510 microprocessor addition instruction. It adds the contents of a given memory location to the contents of the **accumulator**. If the carry flag is set 1 is added to the result. The carry flag should therefore be set to zero with **CLC** when 2 single byte numbers are added together, e.g:

```
LDA $FB
CLC
ADC $FC
```

adds the contents of locations FB and FC, and leaves the result in the accumulator.

After an ADC instruction the carry flag is set to 1 if the result is greater than 255. This allows multi-byte numbers to be added

together with just a few instructions.

| | Status register | N | V | B | D | I | Z | C |
| | | √ | √ | – | – | – | √ | √ |

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	ADC ∗ operand	69	2	2
zero page	ADC operand	65	2	3
zero page, X	ADC operand, X	75	2	4
absolute	ADC operand	6D	3	4
absolute, X	ADC operand, X	7D	3	4*
absolute, Y	ADC operand, Y	79	3	4*
(indirect, X)	ADC (operand, X)	61	2	6
(indirect), Y	ADC (operand), Y	71	2	5*

* Add 1 if page boundary is crossed.

address A number which identifies a location in the computer's memory. Each byte in memory has an address, in the range Ø to 65535. The contents of a particular address can be examined or altered by means of the BASIC keywords PEEK and POKE.
  See **memory map**.

addressing modes The way in which an instruction accesses data. An instruction's addressing mode indicates whether its **operand** is to be treated as data, or the address of data, or as a **vector** to the address of data. Some instructions operate on registers, and have no operands; in which case the operand

and addressing mode is said to be implied. In the 6510 microprocessor there are 9 different addressing modes.

See **absolute addressing; immediate addressing; implied addressing; indexed addressing; indirect addressing; pre-indexed indirect addressing; post-indexed indirect addressing; relative addressing; zero page addressing.**

AND (1) A logical operator which can also act as a bitwise operator. In logical operations, AND tests whether two conditions are true at the same time. It is commonly used with IF. . .THEN, e.g.:

```
IF X>99 AND X<1000 THEN PRINT X;"IS
A THREE DIGIT NUMBER"
```

and can also test for more than two conditions. The following example will only print 'COR-RECT' if all three conditions are true:

```
IF A$=B$ AND M=5 AND K<12 THEN
PRINT "CORRECT"
```

When used as a bitwise operator, AND tests or alters the individual bits in a number. It compares each bit in a number with the equivalent bit in another number. If both are equal to one then it sets the bit in the answer to

one. Otherwise – if one or both bits equal zero – it returns a value of zero. AND is often used in this way to **mask** one or more bits in a number. For example to find out what the bottom four bits in 213 are, AND it with 15. In binary 15 is 00001111. Thus the first four bits of 213 will be ignored since they are being compared with zero:

DECIMAL		BINARY
	213	11010101
AND	15	00001111
	5	00000101

Associated keywords: **OR; NOT**.

(2) A 6510 instruction mnemonic which logically ANDs the contents of a memory location

Status register	N	V	B	D	I	Z	C
	√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	AND # operand	29	2	2
zero page	AND operand	25	2	3
zero page, X	AND operand, X	35	2	4
absolute	AND operand	2D	3	4
absolute, X	AND operand, X	3D	3	4*
absolute, Y	AND operand, Y	39	3	4*
(indirect), X	AND (operand, X)	21	2	6
(indirect), Y	AND (operand), Y	31	2	5*

* Add 1 if page boundary is crossed.

with the contents of the **accumulator**, leaving the result in the accumulator. Can be used to mask bits in the accumulator, e.g. 'AND #$F0' masks off the bottom 4-bits in the accumulator.

See **truth tables**.

argument The number or string which a function operates on. Every function must be followed by an argument enclosed in parentheses, e.g.: SQR(25)
where '25' is the argument.

Among the built-in functions, **FRE** and **POS** take dummy arguments. The value of their arguments is unimportant and can be any number. User-defined functions can also take dummy arguments, e.g.: '10 DEF FN H(N) = INT(X/Y)'.

arithmetic operators The symbols used in arithmetic operations. In the table of operators given below they are listed in order of precedence. This means that some operations are performed before others if their operators have a higher precedence, e.g., the multiplication operator has precedence over the subtraction operator so

9−2×3

equals '3', and not '21'.

Parentheses can be used to override precedences. Thus

$(9-2)\times3$

equals '21'.

Symbol	Use	Example
+	positive number	+5
−	negative number	−8
↑	raise to power of	$2\uparrow3=8$
✕	multiply	$4\times5=20$
/	divide	$12/4=3$
+	add	$7+7=14$
−	subtract	$15-6=9$

array A variable which is used to store sets of data. A number of data items can be assigned to one array and can be identified by their position in the array. This is often a more convenient way of storing data than assigning a variable name to each item. For example a list of names can be stored in a string array as follows:

```
N$(1) = "SMITH"
N$(2) = "TOMKINSON"
N$(3) = "JONES"
N$(4) = "SCOTT"
N$(5) = "COLEMAN"
```

The alternative would be to define a different variable for each name. Not only is it simpler to store a large amount of data in an array but it also makes it easier to manipulate the data. The list above can now be printed out using just three program lines:

```
10  FOR X = 1 TO 5
20  PRINT N$(X)
30  NEXT
```

Arrays can have up to 32768 elements and any number of dimensions. T(5), for example, refers to the fifth element in a one-dimensional numeric array. Two-dimensional arrays can be thought of as arranging their variables in a matrix of rows and columns. Thus $A(4,6) = 6.5$ assigns 6.5 to the sixth item in the fourth row:

A **DIM** statement is required to set up an array. It defines the number of elements and dimensions. The number of elements is counted from zero. So, for example,

```
DIM T(2,1)
```

sets up a numeric array with a total of 6 elements, arranged as follows:

```
T(0,0)  T(0,1)
T(1,0)  T(1,1)
T(2,0)  T(2,1)
```

When a number is used to refer to a particular element it is termed a **subscript**. Attempts to refer to an element outside the range of an array produce a '**BAD SUBSCRIPT**' error message.

ASC A string function which gives the **ASCII** code for a character. It needs to be followed by a string or a string variable between parentheses. If the string has more than one character in it ASC returns the code number of the first character, e.g.: 'PRINT ASC("B")' which prints '66', the ASCII value of the letter 'B'. 'X=ASC("123")' assigns '49', which is the ASCII value for '1', to the variable 'X'.

One of its many uses is to check input from the keyboard. This program asks you to type a number. If the ASCII code of the character you enter is not that of a number it asks you to try again, e.g.:

```
10  PRINT "TYPE A NUMBER FROM 0 TO
    9"
20  GET A$:IF A$=""THEN GOTO 20
30  IF ASC(A$)<48 OR ASC(A$)>57 THEN
    PRINT "TRY AGAIN":GOTO 20
40  PRINT A$
```

Note that a character's ASCII code is not the same as its **screen code**. When a character is displayed on screen of the Commodore 64 the code stored in the **screen memory** is its screen code.

Associated keyword: **CHR$**.

ASCII (American Standard Code for Information Interchange) Before it can store letters or graphics characters in memory the computer needs to represent them as numbers. To do this the Commodore 64 – in common with almost every other microcomputer – uses the ASCII code. It represents each character by a single byte number between 0 and 255. The letter A, for example, is stored in the computer under the ASCII code 65, while the space character is assigned the code number 32.

See **ASC; CHR$**.

Characters, however, are not always represented by their ASCII codes. The Commodore 64 uses its own screen codes to store characters in screen memory. And BASIC keywords are stored as one byte tokens. Note that this version of the ASCII code is not completely standard. The codes for letters, digits, and punctuation are the same as elsewhere but

other codes such as those for certain control characters are unique to the Commodore 64.

See **program area**.

ASCII	CHARACTER	ASCII	CHARACTER
0–4	(not used)	37.	%
5	white	38	&
6–7	(not used)	39	'
8	disable shift Commodore	40	(
9	enable shift Commodore	41	)
		42	✳
10–12	(not used)	43	+
13	return	44	,
14	lower case	45	−
15–16	(not used)	46	.
17	cursor down	47	/
18	reverse-video on	48	0
19	home	49	1
20	delete	50	2
21–27	(not used)	51	3
28	red	52	4
29	cursor right	53	5
30	green	54	6
31	blue	55	7
32	space	56	8
33	!	57	9
34	"	58	:
35	#	59	;
36	$	60	<
		61	=

ASCII	CHARACTER	ASCII	CHARACTER	ASCII	CHARACTER
62	>	91	[	117	◸
63	?	92	£	118	⊠
64	@	93	]	119	◯
65	A	94	↑	120	◆
66	B	95	←	121	▯
67	C	96	▭	122	◆
68	D	97	▲	123	⊞
69	E	98	▯	124	▦
70	F	99	▭	125	▯
71	G	100	▭	126	⊓
72	H	101	▭	127	◣
73	I	102	▭	128	(not used)
74	J	103	▯	129	orange
75	K	104	▯	130–2	(not used)
76	L	105	◪	133	f1
77	M	106	◪	134	f3
78	N	107	◪	135	f5
79	O	108	▭	136	f7
80	P	109	◺	137	f2
81	Q	110	◿	138	f4
82	R	111	▭	139	f6
83	S	112	▭	140	f8
84	T	113	●	141	shifted return
85	U	114	▭	142	upper case
86	V	115	♥	143	(not used)
87	W	116	▮	144	black
88	X			145	cursor up
89	Y				
90	Z				

ASCII	CHARACTER	ASCII	CHARACTER
146	reverse-video off	170	▯
147	clear screen	171	▯
148	insert	172	▯
149	brown	173	▯
150	light red	174	▯
151	gray 1	175	▯
152	gray 2	176	▯
153	light green	177	▯
154	light blue	178	▯
155	gray 3	179	▯
156	purple	180	▯
157	cursor left	181	▯
158	yellow	182	▯
159	cyan	183	▯
160	shifted space	184	▯
161	▮	185	▯
162	▬	186	▯
163	▯	187	▯
164	▯	188	▯
165	▯	189	▯
166	▨	190	▯
167	▯	191	▯
168	▩		
169	◪		

The graphic characters for 192–223 are the same as 96–127.
The graphic characters for 225–254 are the same as 161–190.
Character 255 is the same as 126 and 222.

ASL A 6510 instruction mnemonic which shifts the contents of the **accumulator** or a memory location one bit to the left. Bit 0 is set to 0 and bit 7 moves into the carry flag. It has the effect of multiplying a byte by two, e.g.:

```
LDA #08
STA $C000
ASL $C000
```

loads location C000 with 8 (00001000) and then shifts it to 16 (00010000).

		Status register	N	V	B	D	I	Z	C
			√	–	–	–	–	√	√

addressing mode	assembly language form	op code	No. bytes	No. cycles
accumulator	ASL A	0A	1	2
zero page	ASL operand	06	2	5
zero page, X	ASL operand, X	16	2	6
absolute	ASL operand	0E	3	6
absolute, X	ASL operand, X	1E	3	7

assembler A program which converts **assembly language** instructions into **machine code**. With an assembler you can write machine code programs using **mnemonics** instead of numbers – a much easier task. Assemblers usually allow numbers to be written either in decimal, **hexadecimal**, or **binary**

notation. In addition they allow you to give labels for the destinations of relative and unconditional jump instructions. If you were writing a program directly in machine code, the alternative would be to work out the numerical addresses of the destinations.

The most common format for an assembly language listing is to show three columns of information. Alongside the mnemonic instructions it gives (in hexadecimal) their machine code equivalents and the addresses at which they are stored.

ADDRESSES	MACHINE CODE INSTRUCTIONS		ASSEMBLY LANGUAGE INSTRUCTIONS
19EA	A5 83		LDA $83
19EC	A2 00		LDX #0
19EE	86 96		STX $96
19F0	A2 04		LDX #4
19F2	86 97		STX $97
19F4	A0 00		LDY #0
19F6	91 96	LOOP	STA ($96),Y
19F8	C8		INY
19F9	D0 FB		BNE LOOP
19FB	E6 96		INC $96
19FD	A6 96		LDX $96
19FF	E0 08		CPX #8
1A01	D0 F3		BNE LOOP
1A03	60		RTS

label
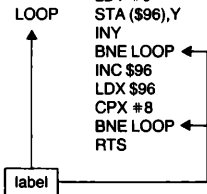
assembly language A language for writing **machine code** programs in which each machine code instruction is represented by a **mnemonic**. Assembly language instructions consists of **operators** and **operands** – the mnemonics themselves and the data or addresses they operate on. The operand part of the instruction also indicates what **addressing mode** the operator is in.

Additionally, assembly language instructions may also contain variables, labels and comments. Labels are the equivalents of **line numbers** in BASIC. Variables, likewise, have the same function as they do in BASIC. Comments are usually preceded by semi-colons and are the same as **REM** statements.

Although numbers can be given in decimal most machine code programmers find it more convenient to use **hexadecimal**.

Before a program written in assembly language can be executed it must be converted into machine code by an assembler. The terms source code and object code refer respectively to the program before and after it has been converted.

ATN A numeric function used to find an

angle whose tangent is already known. The result, the arctangent, is given in radians. It can be converted to degrees by multiplying $180/\pi$. For example, 30 degrees in radians is 0.52359878 and its tangent is 0.57735027

```
PRINT ATN(0.57735027) gives 0.52359878
PRINT ATN(0.57735027)*180/π gives 30
```
Associated keywords: **TAN**; **SIN**; **COS**.

attack/decay The first two phases of a sound **envelope**. In the attack phase, the volume of a note rises from zero to its maximum level, which is set before the envelope is defined. During the decay phase, the volume drops from its maximum to the level set for the sustain phase. Registers 54277, 54284, and 54291 control the attack/decay rates for voices 1, 2 and 3. The duration of the attack and decay phases is determined by the top and bottom four bits in each register. To set these rates, find the value (1st column) corresponding to the desired attack rate, multiply this by 16 add to it the value of the decay rate.

A 500 millisecond attack rate and a 300 millisecond decay rate gives $10 \times 16 + 8 = 168$, so for voice 1 (register 54277):

```
POKE 54277,168
```

VALUE	ATTACK RATE	DECAY/RELEASE RATE
0	2 millisecond	6 millisecond
1	8	24
2	16 ——	48 ——
3	24 ——	72 ——
4	38 ——	114 ——
5	56 ——	168 ——
6	68 ——	204 ——
7	80 ——	240 ——
8	100 ——	300 ——
9	250 ——	750 ——
10	500 ——	1.5 second
11	800 ——	2.4 ——
12	1 second	3 ——
13	3 ——	9 ——
14	5 ——	15 ——
15	8 ——	24 ——

See **sound**; **sustain/release**; **envelope**.

audio/video port Connects the computer to a **monitor** or a hi-fi system. Sending the **SID** chip's sound output to an amplifier instead of a TV loudspeaker generally improves the sound quality.

BAD DATA An **error message**: the program has received a string when it expected a number, e.g. after a **READ**.

BAD SUBSCRIPT An **error message**: the subscript in an array variable is too big.

bank switching A method of giving a

computer more than 64K of memory. 8-bit microprocessors, like the 6510, can only address a maximum of 64K. The Commodore 64, however, has 64K of **RAM** and 20K of **ROM**. It manages the extra memory by switching banks of 4 or 8K in and out of the same address space. Thus, normally the area from 40960 to 49151 is occupied by the BASIC interpreter ROM. But it can be switched out to leave an extra 8K of RAM free for **machine code** programs. Similarly, the **character generator ROM** is switched in and out of the area from 53248 to 57343. This area is also occupied by the **colour memory** and **I/O** RAM.

BASIC (Beginners All-purpose Symbolic Instruction Code) The most commonly used **high-level language** for home computers. On the Commodore 64 a BASIC **interpreter** is built-in and allows programs to be typed in or loaded into **memory** as soon as the machine is switched on.

BASIC extensions Programs which add extra commands to the resident BASIC's set of commands. They are supplied on **cassette** or **disk**, or, like Simon's BASIC, in **cartridge**

form. The extensions usually provide graphics and sound commands, programming **utilities**, and sometimes **structured programming** commands.

BASIC stack An area of **RAM** used by the BASIC **interpreter** to store addresses. When BASIC executes a **GOSUB** instruction it stores its address on the BASIC stack. When it meets a **RETURN** command it removes the address from the stack and branches back to the command after the GOSUB. The addresses enable BASIC to keep track of where it branches from.

A GOSUB command within a subroutine is known as a nested GOSUB.

BASIC handles nested GOSUBs by storing each address in turn on top of the previous address. Like the microprocessor's **stack** it then operates on the last-in, first-out principle and takes addresses from the top of the stack downwards. As the stack is limited to 256 bytes it is possible (but unlikely) for a program to run out of space on the stack by nesting too many GOSUBs. This produces an 'OUT OF MEMORY' **error message**.

Failing to end a subroutine with a RE-

TURN statement also causes this message, and is more common, e.g.:

```
10  GOSUB 20
20  GOTO 10
```

baud The unit of measurement for the rate at which information is transferred from one device to another. Usually it is taken to mean the number of bits passed per second. It is often given as the speed of a computer's cassette storage. On the Commodore 64, programs are saved and loaded at 300 baud.

BCC A 6510 instruction mnemonic which causes a branch if the carry flag is set to 0. This instruction has only one addressing mode, relative addressing. It can branch to any location 129 bytes forward or 126 bytes backward.

Status register   N V B D I Z C
                   – – – – – – –

addressing mode	assembly language form	op code	No. bytes	No. cycles
relative	BCC operand	90	2	2*

* Add 1 if branch occurs to same page.
  Add 2 if branch occurs to different page.

BCS A 6510 instruction mnemonic which causes a branch with **relative addressing** if the carry flag is set to 1,

e.g.:
```
  SEC
  BCS  LABEL
```
sets the carry flag to 1 and causes a branch to
the LABEL address.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
relative	BCS operand	B0	2	2*

* Add 1 if branch occurs to same page.
  Add 2 if branch occurs to next page.

BEQ A 6510 instruction mnemonic which
causes a branch with **relative addressing** if
the zero flag is set to 0. Often used after a
comparison instruction, e.g.:
```
  LDA  $FB
  CMP  #65
  BEQ  LABEL
```
checks whether location FB contains the
ASCII code for the letter A (65). If it does,

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
relative	BEQ operand	F0	2	2*

* Add 1 if branch occurs to same page.
  Add 2 if branch occurs to next page.

then the zero flag is set to 0 and the program branches to the LABEL address.

binary   A number system which uses only two digits, 0 and 1. Since data is stored as a series of binary digits (bits) binary is fundamental to all computer operations. In particular, the **microprocessor** works in binary arithmetic.

Just as the standard number system (decimal) uses ten digits and is said to be to the base 10, so binary numbers are to the base two. This means that the value of each binary digit, passing from right to left, increases by a power of two. In other words each digit is worth two times as much as the digit to its right. To convert a binary number add up its column values, e.g.:

128	64	32	16	8	4	2	1
0	1	0	0	1	1	0	1

$= 64+8+4+1 = 77$

binary coded decimal (BCD)   A way of representing decimal numbers in **machine code**. In BCD form each decimal digit is represented by 4 bits, e.g. 36 in BCD is 00110110: 0011 0110
                                       3   6

By this method a byte can store a number from 0 to 99. When the 6510 microprocessor has been put in decimal mode by a **SED** instruction, it adds and subtracts numbers in BCD form. In binary, 4 bits can store a number from 0 to 15. BCD is therefore less efficient in using memory space. For this reason it is rarely used except in certain business and scientific applications.

bit Each BInary digiT is known as a bit and can either have the value 0 or 1. This is the smallest unit of computer memory. In electronic terms bits can be seen as switches which are either on or off. Most memory operations are performed on **bytes**, but it is sometimes necessary to alter or examine individual bits in a byte. In such cases the bits are referred to by the numbers 0 to 7, with the least significant bit, on the right, being bit 0.

bit No.	7	6	5	4	3	2	1	0
bit value	128	64	32	16	8	4	2	1

BIT A 6510 instruction mnemonic which copies bits 6 and 7 of a specified memory location into the N and V flags in the **status register**. It also performs a logical **AND** with

the contents of the **accumulator** and sets the zero flag accordingly. If the memory byte AND the accumulator equals Ø then the zero flag is set to 1. Note that BIT only alters the status register and not the accumulator or memory byte. It is often used before the branch instructions **BPL** or **BMI**, and **BVS** or **BVC**.

		Status register	N M₇	V M₆	B –	D –	I –	Z √	C –

addressing mode	assembly language form	op code	No. bytes	No. cycles
zero page	BIT operand	24	2	3
absolute	BIT operand	2C	3	4

bit map mode This mode gives **high resolution graphics** with 320 pixels across by 200 down. In **character mode** each character is represented in memory by its code. By contrast, in bit map mode each pixel is represented by a bit. Bit map mode is selected by entering 'POKE 53265,PEEK(53265) OR 32'. This sets bit 5 in the **VIC** chip's control register at 53265 to 1. To turn it off set bit 5 to Ø with 'POKE 53265, PEEK(53265) AND 223'. It is now necessary to tell the VIC chip where the bit map is located. 'POKE 53272,PEEK(53272) OR 8' puts it at 8192. The

area from 8192 to 16191 now serves as the high resolution **screen memory**.

In this mode this POKE also makes the area from 1024 to 2047 act as the bit map **colour memory**. Note that 1024 is normally the start of screen memory in character mode. Each byte in the colour memory controls the colour of a group of pixels in an 8 by 8 character space. Unless **multicolour bit map mode** is selected all the pixels in an 8 by 8 group take the same colour. In this respect the bit map colour memory acts in the same way as the standard colour memory. The difference is that it is also possible to set the background colour for each group of pixels. The lower bits of a byte in colour memory control the colour of the background, and the top 4 bits control the pixel colour. For example, 'POKE 1024,33' colours the pixels in the top left hand character space red, and makes the background white. It does this by giving the lower 4 bits the value 1, the **colour code** for white, and the top 4 bits the value 2, the colour code for red. To set the foreground and background colours for an 8 by 8 block of pixels, use the following formula 'POKE CB, FC✕16 + BC'. 'CB' is the corresponding byte in colour

memory, 'FC' is the colour code for the foreground, and 'BC' is the background colour code.

The BASIC instruction PRINT does not work with high resolution graphics. Nor does the **CLR/HOME key** operate. Instead, to clear the screen, each byte in the screen memory (8192 to 16191) must be set to 0, and the bytes in colour memory (1024 – 2047) must be given the same value. This program switches on bit map mode and then clears the screen, making the background colour red. Any pixels which are set later will appear in white.

```
10  REM TURN ON BIT MAP MODE
20  POKE 53265,PEEK(53265) OR 32
30  REM PUT START OF BIT MAP AT
    8192
40  POKE 53272,PEEK(53272) OR 8
50  FOR N = 1024 TO 2047
60  POKE N,18:REM COLOURS RED AND
    WHITE
70  NEXT
80  FOR N = 8192 TO 16191
90  POKE N,0:REM CLEAR SCREEN
100 NEXT
```

A pixel's position can be described in terms of its X and Y coordinates. The X coordinates

range from left to right, from Ø to 319. Y coordinates from top to bottom run from Ø to 199.

COLUMN Ø	COLUMN 1		COLUMN 39
byte Ø	byte 8		byte 312
byte 1	byte 9		byte 313
byte 2	byte 10		byte 314
byte 3	byte 11		byte 315
byte 4	byte 12		byte 316
byte 5	byte 13		byte 317
byte 6	byte 14		byte 318
byte 7	byte 15		byte 319

ROW 1

| byte 320 |
| byte 321 |
| byte 322 |

ROW 2

BIT MAP SCREEN MEMORY ORGANISATION

Turning on (or plotting) a particular pixel involves setting its corresponding bit to 1. First it is necessary to work out which byte in the bit map (**screen memory**) holds the bit, thus:

$$B = 8192 + 320 \times INT(Y/8) + 8 \times INT(X/8) + (Y \ AND \ 7)$$

where 'X' and 'Y' give the pixel's coordinates and 'B' is the address of the byte required.

This formula is based on the way the screen memory is organised. Each byte has 8 bits which define 8 pixels. The first byte at 8192

defines the first 8 pixels along the top row, the second holds the data for the first 8 pixels in the second row, and so on down to the eighth row. The next block of 8 bytes define the pixels in the next character space along. Thus the ninth byte represents the second row of 8 pixels across the top of the screen.

The bit corresponding to a pixel at a given byte is calculated by 'BIT = 7 −(X AND 7)'. To plot the pixel at location X,Y set the bit to 1 with 'POKE B,PEEK(B) OR (2↑(7−(X AND 7)))' where 'B' is the address of its corresponding byte. To turn off a pixel use 'POKE B,PEEK(B) AND (255 − 2↑(7−(X AND 7)))'.

For a demonstration of high resolution graphics add the following lines to the program above. The resulting program calculates two random points on the screen and then draws a line between them. To stop the program and return to the normal display press RUN/STOP and the **RESTORE key** together.

```
110   X1=RND(0)✳100:X2=RND(0)✳100+219
120   Y1=RND(0)✳200:Y2=RND(0)✳200
130   M=(Y2−Y1)/(X2−X1)
140   C=Y1−X1✳M
```

```
150   FOR X=X1 TO X2
160   Y=X*M+C
170   GOSUB 200
180   NEXT
190   GOTO 110
200   REM PLOT PIXEL AT X,Y
210   B=8192+320*INT(Y/8)+8*INT(X/8)+
      (Y AND 7)
220   POKE B,PEEK(B) OR (2↑(7-(X AND 7)))
230   RETURN
```

BASIC is comparatively slow in handling high resolution graphics. For a faster response use machine code or graphics commands as supplied by BASIC extensions.

bitwise *See* **AND.**

BMI A 6510 instruction mnemonic which causes a branch with **relative addressing** if the negative flag is set to 1; otherwise the next instruction is executed.

Status register	N	V	B	D	I	Z	C
	–	–		–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
relative	BMI operand	30	2	2*

* Add 1 if branch occurs to same page.
 Add 2 if branch occurs to different page.

BNE ´ A 6510 instruction mnemonic which causes a branch with **relative addressing** if the zero flag equals 0. Often used to create a loop, e.g.:

```
        LDX  #25
LABEL   —
        —
        —
        —
        DEX
        BNE  LABEL
```

BNE causes the program to branch back to LABEL 25 times until X=0 and the zero flag is set to 0.

| | Status register | N V B D I Z C |
| | | – – – – – – – |

addressing mode	assembly language form	op code	No. bytes	No. cycles
relative	BNE operand	D0	2	2*

* Add 1 if branch occurs to same page.
  Add 2 if branch occurs to different page.

BPL A 6510 instruction mnemonic which causes a branch with **relative addressing** if the negative flag is set to 0. This instruction is the opposite of **BMI**.

See overleaf.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
relative	BPL operand	10	2	2*

* Add 1 if branch occurs to same page.
 Add 2 if branch occurs to different page.

BREAK An error message: the STOP key has been pressed.

BRK A 6510 instruction mnemonic which forces an **interrupt**. When the program executes a BRK instruction it sets the break flag to 1, pushes the **program counter** (plus one) and the **status register** onto the **stack**, and then passes control to a BRK service routine. The **vector** for this routine is located at 790 and 791.

BRK is often used to provide a debugging facility. By intercepting the BRK vector a programmer can insert a routine to display the contents of the machine's registers. The program can then be interrupted and examined by inserting BRK instructions at selected points. As the value of the program counter is stored on the stack it is easy enough to calculate the return address and resume execution.

	Status register	N	V	B	D	I	Z	C
		–	–	1	–	1	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	BRK	00	1	7

buffer A buffer is a temporary storage area in **RAM** used for holding data when it is transferred from one part of the system to another, e.g. programs are loaded into the computer *via* the cassette buffer.

bug An error in a program either causing it to stop or preventing it from doing what it is intended to do. Most programs in their early stages of development contain bugs. Few people can write a long program that runs perfectly the first time. **Debugging** a program often takes as long as writing it. **Syntax errors** are usually easy to cure, and any bug that produces an **error message** can, generally, be tracked down without too much trouble. The most stubborn bugs are those that do not cause the program to crash.

BVC A 6510 instruction mnemonic which causes a branch with **relative addressing** if the overflow (V) flag is set to 0. This instruc-

tion is used in signed arithmetic operations.
See **two's complement**.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
relative	BVC operand	50	2	2*

* Add 1 if branch occurs to same page.
  Add 2 if branch occurs to different page.

BVS . A 6510 instruction mnemonic which causes a branch with **relative addressing** if the overflow (V) flag is set to 1. Used in signed arithmetic operations when the sign of a number has been changed.
See **two's complement**.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
relative	BVS operand	70	2	2*

* Add 1 if branch occurs to same page.
  Add 2 if branch occurs to different page.

byte The basic unit of computer memory. One byte is made up of 8 bits. 1024 bytes form one kilobyte, usually abbreviated to K.

One byte can hold a number from 0 to 255. It takes 2 bytes to store any number up to

65535. The byte with the higher value is termed the **most significant** byte, while the byte with the lower value is the **least significant** byte. Each byte in the computer's memory has an **address** which specifies its location.

CAN'T CONTINUE An **error message**: the **CONT** command will not work, because either the program has been altered, or it contains an error.

cartridge A plastic box which plugs into the computer and holds a program in **ROM**. Because the program is stored in ROM it is instantly available as soon as the cartridge is plugged in. The program is usually written in machine code and takes over the top 8K of the BASIC **program area**, leaving the user with 30K **RAM** rather than the usual 38K. **Word-processors**, **BASIC extensions**, games, and other languages such as **FORTH** are some of the programs to be found on cartridge. If a program is used regularly this is often a more convenient form of storage than cassette or disk: cartridges are more durable and can be left plugged into the cartridge slot, ready for use when the machine is turned on.

cassette The cheapest form of data storage. Cassette recorders enable the computer to store **programs** and data **files** on cassette tape so that they can be loaded into **RAM** at a later date. Unlike most other home computers the Commodore 64 only takes Commodore's own make of cassette recorder, the Datasette. SAVEing, LOADing, and locating programs on cassette is relatively slow in comparison with other forms of data storage such as **disk**. Unless a program is located by winding the tape forwards or backwards, the computer searches for a program at the same speed at which it loads programs in.

   See **cassette files; fast loading**.

cassette file **Sequential files** are the only type of data files that can be stored on cassette. They are written to tape with the commands **OPEN**, **PRINT#**, and **CLOSE**. Creating a cassette file requires the following steps:
(1) OPEN the file. The third parameter of the OPEN command, the secondary address or command number, is normally 1 to indicate that the file is being written rather than read. A value of 2 means that an end of tape marker is added to the end of the file when it is closed.

Any attempt to read other files after an end of tape marker will cause a 'FILE NOT OPEN' message.

(2) Write to the file using PRINT#.

(3) CLOSE the file. The CLOSE command places an end of file marker at the end of tape. If it is not given data may be lost.

Example:

```
10   OPEN1,1,1,"FILENAME"
20   FOR N=1 TO 10
30   INPUT A$
40   PRINT#1,A$
50   NEXT
60   CLOSE1
```

When a file is read in to memory, the secondary address must be zero. INPUT# reads each record into a variable, GET# reads a character at a time, e.g.:

```
10   OPEN1,1,0,"FILENAME"
20   FOR N=1 TO 10
30   INPUT#1,A$:PRINT A$
40   NEXT
50   CLOSE1
```

This program simply reads records in and prints them on the screen. Usually they are stored in an array.

If the number of records on the file is not

known, the BASIC command **STATUS** can be used to search for the end of file marker, e.g.:

```
20  INPUT#1,A$:PRINT A$
30  IF ST<>64 THEN GOTO 20
```

channel A **sound** output. The **SID** chip has three channels which can produce sounds singly or together. Each channel is controlled by its respective sound registers. Channels are often referred to as **voices**.

character code The code by which a character is represented in memory. In fact the Commodore 64 uses two sets of codes, **screen codes** and **ASCII** codes, but the term character code is generally taken to mean screen code. Each code is a number between 0 and 255 and can be stored in one byte.

character designer A program which allows the user to design characters on an 8 by 8 grid by moving a cursor around with the keys or a **joystick**. Once one or more characters have been designed the program displays the numbers which define each character – to be held in DATA statements. Or it may offer the option of storing the definitions on tape or

disk as a data file. Sprite designers are programs which provide a similar facility for sprites.

character generator ROM   The area of memory where the character definitions are held. When a character is displayed on the screen the **VIC** chip reads its definition from the character generator. As each character occupies an 8 by 8 matrix of pixels, 8 bytes (64 bits) are needed to define one character. 256 definitions for the first character set (upper case/graphics) are stored in ROM from 53248 to 55295; locations 55296 to 57343 contain the definitions for the second set (upper/lower case). The definitions are stored in order of the characters' screen codes. Thus the screen code for the letter B is 2, so the 8 bytes which define it are found from 53248 + 2∗8 onwards. The definition of a character with screen code X starts at

53248 + (8∗X) for character set 1
55296 + (8∗X) for character set 2

Normally, the character generator ROM cannot be accessed from BASIC, and the area from 53248 to 57343 is occupied by RAM. Locations 53248 to 53293, for example, hold

ADDRESS

					BIT PATTERN	**DECIMAL**
	64	16	4	1		
	128	32	8	2		
53256					00011000	24
53257					00111100	60
53258					01100110	102
53259					01111110	126
53260					01100110	102
53261					01100110	102
53262					01100110	102
53263					00000000	0

the VIC registers, but within a program it is possible to switch the ROM in so that characters can be copied into RAM.

See **user defined characters**; **bank switching**.

character mode The normal display mode, in which the screen has 25 rows of 40 **character spaces**. Each space can contain any of the symbols on the keyboard. When the computer is turned on the keyboard gives upper case letters. Pressing the **SHIFT key** displays the **graphics characters** at the right side of the keys, while the graphics characters at the left side of the keys can now be obtained by pressing the Commodore key. If the Commodore key and SHIFT key are pressed the display switches to upper and lower case.

See **character set**; **colour**.

character set The set of all the characters which can appear on the screen. Most of them are available from the keyboard, but a few can only be displayed by POKEing their screen codes into memory.

There are two character sets, but only one can be used at a time. The first gives upper case and **graphics characters**; the second which is switched on by pressing the **Commodore key** and **SHIFT key**, holds upper and lower case and other characters. In addition it is possible to define a new character set. (*See* **user defined characters**). To switch character sets in a program PRINT the required **control character**, e.g. 'PRINT CHR$(14)' selects upper/lower case. 'PRINT CHR$(142)' selects upper case/graphics.

character space An 8 by 8 block of dots or **pixels** – the space occupied by one character.

See **character mode**.

CHR$ A string function – the opposite of the **ASC** function. CHR$ generates the character associated with an **ASCII** code. As its argument it takes a number between 0 and 255, e.g. as the code for the letter A is 65, so

PRINT CHR$(65) prints A on the screen.

CHR$ is used in the following program to print out all the graphic characters on the keyboard.

```
10  FOR N=96 TO 127
20  PRINT CHR$(N);
30  NEXT
40  FOR N=161 TO 191
50  PRINT CHR$(N);
60  NEXT
```

Generally to print a character or string of characters it is simpler to enclose them in quotation marks rather than use CHR$. Thus PRINT "EGG" has the same effect as PRINT CHR$(69); CHR$(71);CHR$(71) but is easier to use. But CHR$ is useful for handling characters that are not easily printed from the keyboard, such as certain **control characters**. This program uses character 14 to switch all the letters on screen to lower case. When a key is pressed it then prints character 142 which switches them back to upper case.

```
10  PRINT CHR$(14)
20  PRINT "TEST"
30  GET A$:IF A$="" THEN 30
40  PRINT CHR$(142)
```

CHR$ also provides a way of testing

whether a **function key** has been pressed.

CLC A 6510 instruction mnemonic which CLears the Carry flag, setting it to 0. It is always used before an **ADC** instruction when two single byte numbers are to be added together. '

CLC can also force a branch, e.g.:

CLC
BCC LABEL

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	0

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	CLC	18	1	2

CLD A 6510 instruction mnemonic which CLears the Decimal flag, setting it to 0. It returns the 6510 microprocessor to binary mode after being in decimal mode.

See **SED**.

	Status register	N	V	B	D	I	Z	C
		–	–	–	0	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	CLD	D8	1	2

CLI A 6510 instruction mnemonic which

enables **IRQ** interrupts by setting the interrupt flag to 0.

See **SEI**.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	0	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	CLI	58	1	2

clock A hardware timing source which synchronises operations within the computer's microprocessor. The rate at which the clock emits pulses is measured in megahertz (MHz). One megahertz equals one million pulses per second. Roughly speaking, the higher a computer's clock rate is, the faster it can process data. The time taken for the microprocessor to carry out a machine code instruction is given in clock pulses or cycles. For example, the instruction **INY** takes 2 cycles. As the Commodore 64's 6510 microprocessor runs at 1 MHz, this instruction takes two millionths of a second to execute.

CLOSE A BASIC input/output statement used to close a **channel** previously OPENed to a peripheral device. For example, when a data file has been written to or read from tape

or disk it must then be CLOSEd. Although not always necessary, it is good practise to CLOSE a channel when it is no longer required.

This command must be followed by the logical file number which identifies a channel, e.g.:

```
CLOSE 3
CLOSE J
CLOSE 2*X
```

Associated keywords: **OPEN**; **CMD**; **GET#**; **INPUT#**; **PRINT#**.

CLR A BASIC statement: on top of the area of memory occupied by a program itself, the computer takes up RAM space to store the program's variables, arrays, and user-defined functions (*See* **DEF FN**). CLR CLeaRs these from memory, but leaves the program untouched. It also frees the RAM occupied by files and the BASIC stack.

A CLR operation is automatically carried out when the RUN command is entered. This means that before numeric variables are assigned a value within the program, the computer gives them a value of zero and makes string variables empty. The following pro-

gram shows the CLR statement in action:

```
10  A=7
20  A$="HELLO"
30  PRINT A,A$
40  CLR
50  PRINT A,A$
```

Associated keywords: **RUN**; **NEW**.

CLR/HOME key Takes the **cursor** to the HOME position at the top left-hand corner of the screen. Pressing the **SHIFT key** and CLR/HOME together homes the cursor and clears the screen at the same time. To perform these actions in a program enter CLR/HOME as a **control character** or use its **ASCII** code with CHR\$. e.g.:

```
10  PRINT "◘"
```

and

```
10  PRINT CHR$(147);
```

have the same effect as pressing SHIFT and CLR/HOME.

```
10  PRINT "◘"
```

and

```
10  PRINT CHR$(19);
```

take the cursor to the HOME position.

CLV A 6510 instruction mnemonic which CLears the oVerflow flag. Only used in signed

arithmetic operations.

See **two's complement**.

	Status register	N	V	B	D	I	Z	C
		–	0	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	CLV	B8	1	2

CMD A BASIC input/output statement. Normally the computer's output is directed to the screen. After a CMD statement, output is redirected to another device. If the device is a printer, PRINT and LIST now send text to the printer instead of displaying it on screen. CMD must be preceded by an **OPEN** statement, and followed by the file number given as the first parameter in the OPEN statement, e.g. 'OPEN 5,4:CMD5:LIST' lists a program to the printer.

With tape or disk the main use for CMD, together with LIST, is in storing a program as a sequence of **ASCII** codes. By contrast the SAVE command stores programs in **token** format. In ASCII format a program can be incorporated as text on a wordprocessor.

To stop the effect of a CMD statement, a **CLOSE** command on its own is not sufficient:

it is necessary to send a blank line to the device using PRINT#, e.g. 'PRINT#5:CLOSE5,4' cancels the action of the CMD statement to the printer given in the example above.

Associated keywords: **OPEN**; **CLOSE**; **PRINT**; **LIST**.

CMP A 6510 instruction mnemonic which CoMPares the contents of a memory location with the contents of the **accumulator** and sets the zero, negative, and carry flags accordingly. CMP works by subtracting the memory location from the accumulator but does not alter their contents, e.g.:

```
LDA #27
CMP $FB
BEQ LABEL
```

loads 27 into the accumulator and then compares it with the contents of FB. If the contents are equal then the zero is set to 1; if the contents of FB are less or greater than 27 the zero flag is set to 0. The carry flag is set to 1 before this operation and is only cleared (C=0) if the contents of memory are greater than 27. Note that other operations such as **LDA** set the zero flag to 1 when the accumulator contains 0; so in this example:

```
CMP  #00
BEQ  LABEL
```
the CMP instruction is not necessary.

| | Status register | N | V | B | D | I | Z | C |
| | | √ | – | – | – | – | √ | √ |

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	CMP # operand	C9	2	2
zero page	CMP operand	C5	2	3
zero page, X	CMP operand, X	D5	2	4
absolute	CMP operand	CD	3	4
absolute, X	CMP operand, X	DD	3	4*
absolute, Y	CMP operand, Y	D9	3	4*
(indirect, X)	CMP (operand, X)	C1	2	6
(indirect), Y	CMP (operand), Y	D1	2	5*

* Add 1 if page boundary is crossed.

colour 16 colours are available. In character mode they are selected in the following ways:

FOREGROUND COLOUR. Each character can take a different colour. Pressing the **CTRL key** together with one of the colour keys along the top row selects one of the first 8 colours. To select one of the second 8 colours press the Commodore key and one of the colour keys. Once a colour has been chosen all future characters are printed in that colour until another colour key is pressed. The current character colour is held in location 646, and it is possible to set a colour by POKEing its

colour code into this location, e.g. 'POKE 646,7' makes the next character printed on the screen yellow.

From within a program colours can also be set by using **colour control characters** or altering the values in colour memory.

COLOUR:	BLACK	WHITE	RED	CYAN	PURPLE	GREEN	BLUE	YELLOW
CODE:	0	1	2	3	4	5	6	7

COLOUR:	ORANGE	BROWN	LIGHT RED	DARK GRAY	MEDIUM GRAY	LIGHT GREEN	LIGHT BLUE	LIGHT GRAY
CODE:	8	9	10	11	12	13	14	15

See **multicolour mode**.

BACKGROUND COLOUR. Location 53281 controls the background colour for the whole screen except the border, e.g. 'POKE 53281,1' uses colour code 1 to make the background white.

In **extended colour mode** the background to each character space can take a different colour.

BORDER COLOUR. This is controlled by register 53280, e.g. 'POKE 53280,4' produces a purple border.

For details of how to set the colour of **high resolution graphics,** *see* **bit map mode.**

colour control characters Used in **PRINT** statements, they have the same effect as pressing the colour keys: they cause any further characters to PRINTed in a given colour.

Inside quotation marks the key presses which normally determine the next character's colour instead produce graphics symbols.

COLOUR	ASCII	KEYBOARD	DISPLAY
black	144	CTRL 1	■
white	5	CTRL 2	E
red	28	CTRL 3	f
cyan	159	CTRL 4	◣
purple	156	CTRL 5	⠿
green	30	CTRL 6	◆
blue	31	CTRL 7	◆
yellow	158	CTRL 8	π
orange	129	C 1	⋀
brown	149	C 2	◸
light red	150	C 3	✕
dark grey	151	C 4	○
medium grey	152	C 5	⠛
light green	153	C 6	▐
light blue	154	C 7	◆
light grey	155	C 8	⠿

These represent the colour control characters and only take effect when the PRINT statement is executed,

e.g.: PRINT "■TEST"

prints the word TEST in yellow. To insert the control character for yellow press the **CTRL** key and 8.

The same characters can be PRINTed by using their ASCII codes with the CHR$ function.

    PRINT CHR$(30)

is equivalent to

    PRINT "◨"

colour memory The area of memory where the colour codes of characters on screen are stored. In **character mode** it runs from addresses 55296 to 56295. Unlike **screen memory** colour memory cannot be moved elsewhere.

The colour of characters on screen can be set by POKEing colour codes into their corresponding locations in colour memory. When a character has already been PRINTed to the screen its colour can be changed in this way. Otherwise the PRINT statement assigns the current foreground colour, irrespective of the values in colour memory.

When characters are displayed by POKEing their **screen codes** into the screen memory it

is essential to set the corresponding locations in colour memory. If this is not done they take the same colour as the background and are not visible. As an example, the following program displays character set 1 in each of the 16 colours in turn.

```
10  PRINT CHR$(147)
20  FOR N=0 TO 255
30  POKE 1024+N,N
40  NEXT
50  FOR C=0 TO 15
60  FOR N=0 TO 255
70  POKE 55296+N,C
80  NEXT
90  NEXT
```

In **bit map mode** the normal screen memory acts as the colour memory and the area from 55296 to 56295 is not used.

Commodore key The key at the left of the keyboard which bears the Commodore symbol. It has three functions: when pressed with the **SHIFT** key it switches between the two **character sets**; when held down while pressing a colour key it gives the colours with codes from 8 to 15; when held down while pressing a graphics key it produces the left-

hand **graphics character**.

compiler A program which converts a program written in a **high level language** into machine code. Normally, a BASIC program is translated into machine code by the computer's BASIC **interpreter**, but the interpreter only operates while the program is running, converting one instruction at a time. Consequently, interpreted BASIC is comparatively slow. By contrast, a compiler converts a program into machine code before it is executed. Once compiled a program can then be run or saved as machine code on cassette or disk. Compiled programs typically run at least ten times faster than normal BASIC programs.

composite video A type of video signal which allows a **monitor** to be used instead of a television. Normally the video output is fed through a modulator which raises the frequency of the signal so that it is equivalent to a UHF TV signal. Because composite video signals are not modulated but sent directly to the monitor they give a higher quality picture.
    See **audio/video port**.

CONT A command used to restart (CON-

Tinue) a program after it has been halted. A
STOP or an **END** statement will halt a
program while it is running, as will pressing
the **RUN/STOP key**. CONT may then be
entered as a direct command to resume execu-
tion at the next statement. Its principal use is in
debugging programs. To locate an error in a
program it is often a good idea to insert STOP
statements at various stages. When the pro-
gram halts you can examine its progress by
printing out crucial variables. Then you can
re-start by typing CONT. If you edit a line, or
the program has been halted by an error,
CONT will not work. Attempts to resume
program execution will produce the error
message 'CAN'T CONTINUE'.

The following program simply adds up all
the numbers from 1 to 1000. Press the RUN/
STOP key while it is running. By typing
'PRINT N,TT' you will be able to see how far
it has got. Then type CONT to continue.

```
 5  TT=0
10  FOR N=1 TO 1000
20  TT=TT+N
30  NEXT
40  PRINT "THE TOTAL IS ";TT
```

Associated keywords: **END**; **STOP**.

control character Characters which control output to the screen or any other device. When they are **PRINT**ed they do not appear on screen but reproduce the effect of pressing their corresponding keys, e.g. '10 PRINT CHR$(14)' is equivalent to pressing the **RUN/STOP** and **SHIFT** keys. It switches the character set to upper/lower case.

Some control characters can appear in **strings** between quotation marks. They are represented by graphics characters. (*See* **colour control characters**; **cursor control characters**; **reverse characters**). Like any other characters they can be assigned to string variables, e.g.:

```
10 A$ = CHR$(20)+CHR$(20)
20 PRINT "TWIST"A$"N"
```

prints

    TWIN

20 is the ASCII code for the delete character; here it has the same effect as pressing the **INS/DEL** key twice.

COS A function which calculates the cosine of an angle. The angle must be given in radians. To convert an angle in degrees (Y) to radians multiply it by $\pi/180$, e.g.:

10   X=COS(Y✶π/180)
Associated keywords: **SIN**; **ATN**.

CP/M (Control Program for Microprocessors)
A **disk operating system** which allows a
wide range of business and application soft-
ware to be run. CP/M only works with Z80
microprocessors. On the Commodore 64 it
requires a CP/M cartridge which contains a
Z80 chip.

CPX A 6510 instruction mnemonic which
ComPares the contents of X **index register**
with the contents of memory. This instruction
acts in the same way as **CMP** except that the
memory byte is subtracted from the X register
instead of the **accumulator**.

		Status register	N	V	B	D	I	Z	C
			√	–	–	–	–	√	√

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	CPX ✳ operand	E0	2	2
zero page	CPX operand	E4	2	3
absolute	CPX operand	EC	3	4

It is often used in a loop, e.g.:

```
        LDX  #200
  LAB1  DEX
        —
```

```
        —
        CPX  #100
        BNE  LAB1
```

CPY A 6510 instruction mnemonic which ComPares the contents of the Y **index register** with the contents of memory. It acts in the same way as **CMP** except that the memory byte is subtracted from the Y register instead of the **accumulator**. Commonly used to control a loop.

	Status register	N	V	B	D	I	Z	C
		√	–	–	–	–	√	√

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	CPY # operand	C0	2	2
zero page	CPY operand	C4	2	3
absolute	CPY operand	CC	3	4

CTRL key Selects the first 8 colours when pressed with keys 1 to 8. CTRL plus key 9 (RVS ON) gives reverse characters; CTRL and key 0 (RVS OFF) turns them off. Holding down CTRL after a LIST command slows down the rate at which program lines are displayed on screen.

  See **reverse characters**.

cursor The flashing white square which

indicates where the next character typed at the keyboard will appear on the screen. When a program is running the cursor disappears. Its reappearance beneath the READY message shows that a program has ended and that the computer will respond to commands.

See **cursor keys**.

cursor control characters When the cursor keys are pressed between quotation marks instead of moving the cursor they produce **control characters**. These are shown as graphics symbols, and only take effect when a PRINT statement is executed.

CURSOR KEY	SHIFT KEY	ASCII CODE	GRAPHICS CHARACTERS
CLR	√	147	
HOME		19	
cursor up	√	145	
cursor down	→	17	
cursor right		29	
cursor left	√	157	

For example:
    10 PRINT ""
sends the cursor to the HOME position and clears the screen. In other words, within a program it is equivalent to pressing the **CLR/HOME** and **SHIFT** keys. Another way of PRINTing these characters is to use their

ASCII codes with the **CHR$** function, e.g. '10 PRINT CHR$(147)' has the same effect as the line above.

Cursor control characters are useful for fixing the position at which characters are printed.

```
10  PRINT "A 🄾 🄾 🄾 🄾 🄳 🄳 B"
```

PRINTs B four rows down from A, two spaces along.

cursor keys Primarily used in editing a program line, these keys can take the cursor to any position on the screen. Any character typed at the keyboard will appear at the new position. (*See* **screen editor**). Pressing the up/down and right/left keys moves the cursor up and to the right. When the SHIFT key is held down these keys move the cursor down and to the left. If the cursor keys are pressed between quotation marks they produce **cursor control characters**.

DATA A BASIC statement which when used in conjunction with the **READ** statement, holds numeric or string data which will be needed by a program. The data may either be assigned to variables of arrays, or used immediately in the program. For example, one

way of storing a tune is to put the values of the notes in DATA statements and read them in to be played.

Each item in a list of data following a DATA statement must be separated by a comma. If the list contains two consecutive commas the computer will interpret this as a zero or an empty string. Normally string items do not need to be enclosed in quotes unless the string includes commas, colons, graphics characters, leading or trailing spaces. DATA statements can be placed anywhere in a program but they are usually put together at the end. Here they are used to hold the months of the year and the number of days in each:

```
10  FOR N=1 TO 12
20  READ M$,D
30  PRINT M$;" HAS ";D;" DAYS"
40  NEXT
50  DATA  JANUARY,31,FEBRUARY,29,
    MARCH,31
60  DATA  APRIL,30,MAY,31,JUNE,30
70  DATA  JULY,31,AUGUST,31,SEPTEMBER,30
80  DATA  OCTOBER,31,NOVEMBER,30,
    DECEMBER,31
```

Note that the string data items are read into a string variable, M$, and the numeric items

into a numeric variable, D. Attempts to assign data items to the wrong variable type are a common cause of error, and will produce the message '?SYNTAX ERROR'.

Associated keywords: **READ**; **RESTORE**.

data base A computerised filing system. Database programs allow large amounts of data to be organised in a file in RAM, and then saved on cassette or disk. Each file is usually made up of records which hold sets of related data. For example, a single record could contain a name, address, and telephone number. By means of a database, data can be entered or deleted, formatted, sorted, and – most importantly – accessed rapidly.

debugging The process of finding and eliminating a **bug**. If a bug brings a program to a halt the resulting **error message** usually gives a clue to its whereabouts. But note that the error is not necessarily in the line referred to in the message, but may be in an earlier line. Otherwise, a 'TRACE' **utility** is a useful aid to debugging. It prints out the numbers of program lines as they are executed, enabling the user to see which parts of the program are working correctly. Another technique is to

insert **STOP** statements at crucial points in the program. When it halts the user can examine the contents of selected variables by PRINT-ing them out as a **direct command**. In this way the location of a bug can be narrowed down to one section of the program.

DEC A 6510 instruction mnemonic which DECreases the contents of a memory location by one, and sets the zero flag to 1 if the result is 0, e.g. 'DEC $D864' decreases the byte at D864.

| | Status register | N | V | B | D | I | Z | C |
| | | √ | – | – | – | – | √ | – |

addressing mode	assembly language form	op code	No. bytes	No. cycles
zero page	DEC operand	C6	2	5
zero page, X	DEC operand, X	D6	2	6
absolute	DEC operand	CE	3	6
absolute, X	DEC operand, X	DE	3	7

DEF FN A statement used to create a user-defined function which can be called later in the program by the keyword **FN**. If the same formula is to be used in several different places in a program it is convenient to assign it to a user-defined function. Unlike some versions of BASIC, on the Commodore 64 you can only define a mathematical function: DEF FN

will not handle string functions.

It takes the form 'DEF FN F(X)' where 'F' is the name of the function and 'X' is a variable. The variable does not need to be included in the formula. In the second of these two examples the variable B is not used in the function:

```
DEF FN A(C)=9×C/5+32
DEF FN A(B)=SQR(X×X+Y×Y)
```

The first function converts temperature given in centigrade to fahrenheit by operating on the variable C, but in the second the value of the variable B has no effect on the result.

The following program defines a function to work out the decimal fraction part of a number:

```
10  DEF FN F(X)=X−INT(X)
20  PRINT FN F(3.75)
30  PRINT FN F(12/5)
40  A=7:B=3:X=A/B
50  IF FN F(X)<>0 THEN PRINT B;" IS
    NOT A FACTOR OF ";A
```

Note that although a DEF FN statement can be placed anywhere in a program, it must occur before its corresponding FN statement is first used.

Associated keyword: **FN**.

DEVICE NOT PRESENT An error message: an **I/O** device such as a printer or disk drive has not been connected.

DEX A 6510 instruction mnemonic which DEcreases the contents of the X **index register** by one, and sets the zero flag to 1 if the result is 0. Often used with **indexed addressing** and to decrease the value of X when it acts as a loop counter.

Status register	N	V	B	D	I	Z	C
	√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	DEX	CA	1	2

DEY A 6510 instruction mnemonic which DEcreases the contents of the Y **index register** by one, and set the zero flag to 1 if the result is 0. Often used with indexed addressing (*see* **addressing modes**) and to decrease the value of Y when it acts as a loop counter.

Status register	N	V	B	D	I	Z	C
	√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	DEY	88	1	2

DIM A statement. Before an **array** can be

used in a program it needs to have been set up by a DIMension statement. It tells the computer how many dimensions the array has and how many elements there are in each. It takes the form 'DIM variable (integer, integer,. .)', e.g.:

```
DIM C$(5,6)
```

The variable 'C$' gives the array a name and indicates what type of array it is. In this case it is a string array and will only accept strings. Other types are integer or numeric arrays. The integers, 5 and 6, specify the number of elements in each dimension. As the elements are numbered from 0 onwards there is always one more in each dimension than is specified, e.g.: 'DIM A(20)' defines a numeric array with one dimension containing 21 elements. 'DIM B$(3,6)' defines a 4 by 7 string array. 'DIM N%(10,2,15)' defines an integer array with three dimensions, 11 by 3 by 16.

A single DIM statement can be used to set up more than one array.

```
10   DIM A$(9),B$(3,5),T(8)
```

is equivalent to

```
10   DIM A$(9)
20   DIM B$(3,5)
30   DIM T(8)
```

A DIM statement may only be executed once if a program. Executing it twice will cause a 'REDIM'D ARRAY' error. If you do not DIM an array before using it the computer will assume it has 11 elements.

direct command   As direct commands, single keywords or lines of BASIC can be entered from the keyboard and executed immediately by pressing the **RETURN key**. Although the command may remain on the screen the computer does not store it in memory after it has been executed. By contrast, program commands, which are preceded by line numbers, are stored in memory when the RETURN key is pressed.

One of the many uses for direct commands is to make the computer serve as a calculator. e.g.:

```
PRINT 3.5×9 + 42
```

It is also possible to enter multi-statement lines directly:

```
FOR N = 1 TO 1000:PRINT N:NEXT
```

disassembler   A program for converting **machine code** into **assembly language**. By substituting mnemonics for numerical instructions disassemblers make machine code pro-

grams easier to follow. They are often used to examine the computer's built-in programs in ROM such as the BASIC interpreter or the operating system.

disk   *See* floppy disk.

disk commands   On top of the BASIC commands for handling data such as SAVE and GET#, the disk operating system (**DOS**) supplies its own commands. These fall into two groups, disk maintenance commands and disk utility commands. The first group is as follows:

NEW	formats a disk.
RENAME	renames a file.
COPY	copies a file.
SCRATCH	erases a file.
VALIDATE	re-organises the files on disk to make more space available.
INITIALIZE	prepares a disk for use.
LOAD "$"	loads the directory which can then be listed.

Apart from 'LOAD "$"' commands of this sort are given as command strings after a PRINT# statement, e.g.:

```
OPEN 1,8,15
PRINT#1,"SCRATCH:PROG1"
```

erases PROG1

´The disk utility commands are documented in the Commodore 1541 disk drive User's Manual. They include commands for creating **relative files** and using the disk drive with **machine code**.

disk drive   A faster and more flexible way of storing data than **cassette**. Whereas it may take over 10 minutes to SAVE or LOAD a program on tape, the same process can be completed in a matter of seconds on disk. In addition to this, the disk drive can access data rapidly at any part of the disk. The other advantage of disk over cassette is that the disk operating system (**DOS**) supervises the way programs are stored on disk, thus saving the user the trouble of locating programs. When a program is saved the DOS finds space for it on the disk and records its name in the disk directory. By LISTing the directory the user can see what programs are stored and how much space is left. The DOS also provides a set of **disk commands** for manipulating files on disk. They include commands to rename a file, erase it, or copy it.

Only the Commodore disk drive, the 1541,

can be connected to the Commodore 64. Other drives require an **interface**. Up to 4 disk drives can be linked up to the computer in a 'daisy-chain' arrangement via the **serial port**. The type of disk used is a **floppy disk**.
  See **sequential files; relative files**.

displacement A 1-byte number following a branch instruction which indicates how far backward or forward the program should branch.
  See **indexed addressing**.

display mode The way in which characters or graphics are displayed on screen. When the computer is turned on it is in **character mode**. Within this mode **multicolour mode** and **extended colour mode** can be selected as options. **Bit map mode** and **multicolour bit map mode** allow **high-resolution graphics**.

DIVISION BY ZERO An **error message**: this is not allowed. It is usually caused by dividing a number by a variable.

DOS (disk operating system). The program that controls and supervises data storage on disk, and provides a range of different **disk commands**. Unlike most other **disk drives**,

the Commodore 1541 disk drive contains its own DOS in 16K of **ROM** together with 2K of **RAM** and a 6502 microprocessor. Whenever a program or data file is stored, the DOS records details of which tracks and sectors have been used in the BAM (Block Availability Map) which is held at track 18. In this way it can calculate how much space is available. The DOS also keeps a list of the names of the files on a disk in a directory, which is also stored at track 18.

empty string A string variable with no characters in it. Use two quotation marks to empty a string variable, e.g.:
```
10  A$ = ""
10  IF A$="" THEN ....
```

END A BASIC statement. When the computer meets an END statement in a program it stops running the program and returns control to the user. The only difference between END and **STOP** is that whereas STOP indicates the line at which the program has halted END simply displays the READY message. A program will finish when the last line as been executed, so it is not necessary to put an END statement at the end. Within a program it may

be used any number of times, as required.
Here it stops the program if 'NO' is entered:

```
100   PRINT "DO YOU WANT TO PLAY
      AGAIN?"
110   INPUT A$
120   IF A$="NO" THEN END
130   GOTO 100
---
---
999   END
```

Associated keywords: **STOP; CONT.**

envelope   Determines the way a note rises
and falls in volume. An envelope has four
phases, attack, decay, sustain and release
(ADSR) and is defined by POKEing the
attack/decay and **sustain/release** sound reg-
isters. Each type of sound has a characteristic
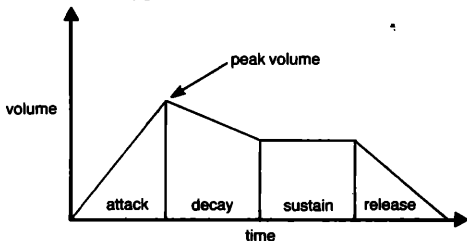
envelope shape and **waveform**. For example, a piano sound rises sharply and then decays more slowly, while an organ has fast attack and decay phases but a prolonged sustain level.

EOR A 6510 instruction mnemonic which performs an Exclusive OR operation between the contents of the **accumulator** and the contents of a memory location, leaving the result in the accumulator. In an EOR operation the corresponding bits in two bytes are compared. If one bit is 0 and the other 1 then the result is 1; otherwise the result is zero.

Status register		N	V	B	D	I	Z	C
		√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	EOR # operand	49	2	2
zero page	EOR operand	45	2	3
zero page, X	EOR operand, X	55	2	4
absolute	EOR operand	4D	3	4
absolute, X	EOR operand, X	5D	3	4*
absolute, Y	EOR operand, Y	59	3	4*
(indirect, X)	EOR (operand, X)	41	2	6
(indirect), Y	EOR (operand), Y	51	2	5*

* Add 1 if page boundary is crossed.

EORing a byte with 255 (11111111) has the effect of inverting (or flipping) its bits. Inverting a byte gives its complement,

e.g.:

Decimal	Binary
181	10110101
EOR 255	11111111
74	01001010

error message   A message produced by the computer, indicating a program error, e.g.: '10 POKE 1024,300' results in 'ILLEGAL QUANTITY ERROR IN LINE 10' since a location cannot be POKEd with a value greater than 255.

EXP   A numeric function that calculates 'e' (2.71828183) raised to a given power. For example, 'EXP(3)' returns 20.855369, the value of 'e' cubed.

Associated keyword: **LOG**.

expansion port   Also known as the cartridge slot, this is a 44-pin edge connector. It gives access to the Commodore 64's main address and data lines, thus providing a large measure of control over the computer's functioning and memory configurations. Generally it is used to take programs in **ROM**, such as games, or to connect interfaces to a variety of devices, e.g. **light pens** and **speech synthesisers**.

expression A combination of numbers, strings, or variables with logical or arithmetic operators, e.g.:

```
A <> B
(A=5) AND (B=6)
```

Expressions are mainly used in IF. . .THEN statements, e.g.: 'IF A$ = "NAME" THEN. . .' where A$ = "NAME" is an expression.

extended colour mode A display mode which allows different character spaces to have different backgrounds. In the standard **character mode** the screen takes the same background colour throughout. In extended colour mode each space can take one of four background colours. Bits 6 and 7 of the corresponding bytes in the **screen memory** are used to hold the colour information. This leaves only 6 bits for the **screen code** of a character. As a result, in this mode only the first 64 characters in the character set can be displayed. Note that these are the characters associated with the first 64 screen codes, not the **ASCII** codes.

When a character whose code is greater than 63 is POKEd to the screen it is converted to

one of the first 64 characters. The top two bits of its code are ignored and serve instead to select the background colour. For example POKEing code 66 into screen memory displays the letter B, whose code is 2, with background colour 1. Normally 66 is the code for a graphics character. The background colours are selected by POKEing colour codes into registers 53281 to 53284. The following table shows which how to set each of the four backgrounds. Note that characters whose code is less than 64 take the normal screen colour.

CHARACTER CODES	BACKGROUND COLOUR NUMBER	BACKGROUND COLOUR REGISTER
0–63	0	53281
64–127	1	53282
128–191	2	53283
192–255	3	53284

Extended colour mode is controlled by bit 6 in register 17 (53265) of the VIC chip. 'POKE 53265,PEEK(53265) OR 64' turns it on. 'POKE 53265,PEEK(53265) AND 191' turns it off.

EXTRA IGNORED An error message: too many items have been entered in response to an INPUT prompt.

fast loader A program that speeds up the rate at which the Commodore 64 loads and saves programs. Normally the computer saves programs at 300 **baud**. By the standard of many other home computers this a comparatively slow rate. Long programs can take over 10 minutes to load. (The computer stores each program twice so that it can check for errors when loading it back to **RAM**; dispensing with this precaution is enough to double the loading rate.) Fast loaders load programs up to 8 times faster than normal. They work by copying the computer's cassette filing routines from **ROM** into RAM and then modifying them; or by replacing them entirely.

Commercially sold software often includes a short **machine code** routine at the front of the tape to fast-load the main program. Programs are also available for speeding up the Commodore disk drive's loading rate. Again, the Commodore disk drive is substantially slower than drives used with other computers.

file A set of data stored on cassette or disk. The word data is used here in the widest sense to include programs as well as the data they work on. Thus files are sometimes divided

into program files and data files. There are two
types of data file, **sequential** and **relative
files**. Relative files cannot be held on cassette.
Data files generally store sets of related data.
E.g. a list of addresses or a set of figures.
    See **database**.

FILE NOT FOUND   An **error message**: an
attempt has been made either to read a file after
an END OF TAPE marker, or to load a
non-existent file from disk.

FILE NOT OPEN   An **error message**:
the OPEN command has not been given pre-
viously.

FILE OPEN   An error message: the file has
already been OPENed.

filter   Used to suppress or attenuate certain
sound **frequencies** above or below a cut-off
point. Four registers control the filters. Regis-
ters 54293 and 54294 hold the cut-off frequen-
cy value; register 54295 determines which
voices are to be filtered; and register 54296
selects the type of filter. There are three types:
high-pass, low-pass and band-pass filters.
    See opposite.

register 54293	
BIT	FUNCTION:
0–2	filter cutoff value (least significant bits)
3–7	Not used

register 54294	
BIT	FUNCTION:
0–7	filter cutoff value (most significant bits)

register 54295	
BIT	FUNCTIONS:
0	filter voice 1
1	filter voice 2
2	filter voice 3
3	filter
4–7	resonance

register 54296	
BIT.	FUNCTIONS:
0–3	volume
4	low pass filter
5	band pass filter
6	high pass filter
7	turn off voice 3

flag Indicates whether an event has or has

not occurred. Flags are generally represented by single bits in memory or in a register, and take a value of either 1 or 0.

See **status register**.

floating point variables Store whole and fractional numbers, and are accurate up to nine digits, e.g.:

$$TT = 9.88$$
$$N = -0.06$$
$$F2 = 25$$

Numbers larger than 999999999 or less than 0.01 are displayed in scientific notation, e.g.: 3567000000000 is converted to 3.567E+12. In this form numbers are expressed as the product of their exponents and a number between 1 and 10, e.g.:

$$2500 = 2.5 \times 10^3 = 2.5E+3$$
$$0.0000756 = 7.56 \times 10^{-5} = 7.56E-5$$

floppy disk The type of disk used by the Commodore **disk drive**. As on many other home computers, the Commodore drive takes 5.25-inch soft-sectored, single-sided, disks. Each disk has a storage capacity of almost 170K. Information is stored in concentric circles known as tracks. Each track is divided into sectors which hold blocks of 256 bytes. Before

a disk can be used it needs to be formatted. In formatting a disk the disk operating system defines the tracks and sectors it is going to use. The Commodore disk format has 35 tracks with 17 to 21 sectors.

FN A numeric function used to call a function which has already been defined by a **DEF FN** statement. It must be followed by the name of the function, and a number or numeric variable in parentheses, e.g. 'DEF FN A(X)=(X−32)/9∗5' defines function A which converts Fahrenheit temperatures to centigrade. Here are some of the ways it could be used:

```
10  PRINT FNA(66)
10  C=FNA(120)
10  IF FNA(X)=100 THEN PRINT "BOILING"
```

Associated keyword: **DEF FN**.

FOR A statement, used together with **TO** and **NEXT**, which tells the computer to repeat an action a given number of times: it sets up a loop. The statements which are to be repeated are those between FOR and NEXT. In this example a FOR. . .NEXT loop is used to print the word TEST five times:

```
10  FOR T=1 TO 5
```

```
20  PRINT "TEST"
30  NEXT
```

FOR requires you to specify the following elements:

a numeric variable to act as a loop counter

an initial value for the counter

a limiting value

In the example above, the variable 'T' acts as the loop counter. Initially 'T' is set to '1'. When the program reaches 'NEXT' it increases 'T' by one, until 'T' equals '5'. Then it passes on to the first statement after 'NEXT'.

You can use the value of the loop counter within the loop itself, as in this program which prints the numbers 200 to 300:

```
10  FOR N=200 TO 300
20  PRINT N
30  NEXT
```

It is also possible to supply variables for the initial and limiting values of the counter. This program performs the same action as the one above, using variables:

```
10  S=200:F=300
20  FOR N=S TO F
30  PRINT N
40  NEXT
```

If one FOR. . .NEXT loop is contained in

another it is known as a nested loop. Here two loops are used to print out the multiplication tables:

```
10   FOR N=1 TO 12
20   FOR T=1 TO 12
30   PRINT N;" X ";T;" = ";N✳T
40   NEXT
50   NEXT
```

Normally the loop counter is increased by one. By including the STEP statement you can specify the size of the increment.

See **STEP**.

Associated keywords: **NEXT**; **STEP**; **TO**.

FORMULA TOO COMPLEX An error message: a string or an arithmetic expression is too complex and should be split into two parts.

FORTH An alternative **high-level language** to BASIC. Originally devised for controlling external devices, FORTH is now used for more general purposes. One of its unusual features is that it allows the user to add new keywords – called words in FORTH – to its dictionary. New words are defined as a sequence of existing words. Not only is FORTH more flexible than BASIC but it is also much

faster: each word is compiled before it enters the dictionary. FORTH is available for the Commodore 64 on cartridge.

See **compiler**.

FRE A function which returns the number of bytes in memory which are unused and free for your program. It takes the form FRE(X) where the value of X is unimportant and can be any number. Sometimes FRE returns a negative result, in which case add 65536 to find the actual number of unused bytes.

'FRE(0)−(FRE(0)<0)✱65536' always gives the correct positive result. FRE(0) is useful for finding out how much space you have left for a program and its variables, or for working out its length. When the machine is turned on, 38911 bytes are available to the user. The following program takes up 48 of them. Line 30 prints out the length of the program.

```
10  REM
20  REM
30  PRINT 38911-(FRE(0)-
    (FRE(0)<0)✱65536)
```

frequency Determines the pitch of a sound: the higher the frequency, the higher the sound. Each **voice** has a high and a low byte

frequency register. To convert a frequency value into two bytes use:

   FL=INT(F/256):FH=F-FL×256

where 'F' is the frequency and 'LF' and 'HF' are its low and high values, e.g. 'POKE 54272,135:POKE 54273,33' sets the pitch of voice 1 to middle C. This note has a frequency number of 8583, which equals 256×33 + 135.

	FREQUENCY LOW BYTE	FREQUENCY HIGH BYTE
voice 1	54272	54273
voice 2	54279	54280
voice 3	54286	54287

   See **music note values**.

function   A BASIC instruction which performs a calculation on a number or a string. The **argument** of a function must be enclosed in parentheses, e.g.:

   L = SQR(55)
   A$ = LEFT$("WEDNESDAY",3)

function key   The four keys at the right of the keyboard marked f1 to f7. Each function key has an associated CHR$ code which allows its keystroke to be tested. Otherwise, they have no effect. Holding down the **SHIFT key** while a function key is pressed gives four

more testable keystrokes from f2 to f8.

When a function key is pressed between quotation marks it produces a graphics character. This provides an alternative way of checking for its keystroke, e.g.:

```
10   GET A$:IF A$=CHR$(133) THEN. . .
```

or

```
10   GET A$:IF A$="□" THEN. . .
```

On some other computers the function keys can be programmed to produce a string of commands, as if they were entered from the keyboard. This is also possible on the Commodore 64, using a machine code routine.

GET A statement which reads a character from the keyboard into a variable. It must be followed by either a string or numeric variable. GET M$ and GET N are examples of each. If the variable is numeric it expects a number. Pressing a non-numeric key will then produce a '?SYNTAX ERROR' message. GET is similar to INPUT except that it does not wait for RETURN to be pressed, and does not display the character it picks up on the screen. If fact, it does not wait for a keystroke, and if no key is pressed it assigns zero to a numeric variable or the **empty string** to a

string variable.

If you want GET to wait until a key is pressed you need to place it in a loop, as in line 10 here:

```
10 GET A$:IF A$="" GOTO 10
20 PRINT A$
```

Strictly speaking, GET does not read the keyboard but the **keyboard buffer**. As the buffer stores keystrokes, GET may return a character even though no key is pressed. It is sometimes necessary to clear the keyboard buffer before using GET.

If GET is used to input numeric data it is often preferable to assign the data to a string rather than a numeric variable. By doing this you avoid the risk of crashing the program by pressing a non-numeric key. The following input routine only accepts numbers. Note that it checks whether the RETURN key has been pressed by looking for its associated character code, CHR$(13). Then it converts the string data in B$ to numeric form using the VAL function.

```
10 GET A$:IF A$="" GOTO 10
20 IF A$=CHR$(13) THEN GOTO 60
30 IF A$<"0" OR A$>"9" THEN GOTO 10
40 PRINT A$;:B$=B$+A$
```

```
50  A$="":GOTO 10
60  N=VAL(B$)
```
Associated keyword: **INPUT**.

GET# An input/output statement that works in the same way as GET except that it inputs data from a peripheral device rather than the keyboard. It is primarily used for reading one character at a time from a data file on tape or disk. GET# needs to be followed by a logical file-number and a variable, as in 'GET#2,A$' or 'GET#1,N'. The file number directs GET# to a particular device and must have been previously specified in an OPEN statement. In this program GET# reads 20 characters from a sequential file on tape and displays them on screen:

```
10  OPEN 2,1,1
20  FOR T=1 to 20
30  GET#2,A$
40  PRINT A$
50  NEXT
60  CLOSE 2
```

If the device number in the OPEN statement is 3, GET# reads characters from the screen. It can be used to dump a copy of the screen to the printer.

Associated keywords: **CLOSE**; **INPUT#**; **PRINT#**; **OPEN**.

GOSUB A statement. Like the **GOTO** statement, this command (short for GO to a SUBroutine) transfers control to a different part of the program: it causes the program to branch to the line number following the GOSUB statement. But unlike GOTO, it remembers where it branched from. When the program meets a **RETURN** statement it jumps back to the first statement after the original GOSUB.

In the following example, the subroutine starting at line 100 is called three times and is used to calculate the length of the word stored in 'A$'. Note that the END statement at line 70 is necessary to prevent the program running on to line 100.

```
  10   A$="FLOWER"
  20   GOSUB 100
  30   A$="IMMEDIATELY"
  40   GOSUB 100
  50   A$="CONSTANTINOPLE"
  60   GOSUB 100
  70   END
 100   L=LEN$(A$)
```

```
110 PRINT A$;" HAS ";L;" LETTERS
    IN IT"
120 RETURN
```
Associated keywords: **ON**; **RETURN**.

GOTO A statement. Normally when the computer has executed a statement it then proceeds to the next one. GOTO followed by a line number causes it to jump to the line specified which may be elsewhere in the program. It can also be followed by a variable. 'A=200: GOTO A' has the same effect as 'GOTO 200'.

It is often used for skipping one or two lines if a particular condition is not met, as in:
```
10 PRINT "HAVE YOU HAD ENOUGH?"
20 INPUT A$
30 IF LEFT$(A$,1)="Y" THEN GOTO 50
40 GOTO 10
50 PRINT "GOODBYE"
```
Within a program, GOTO, like GOSUB, changes the order in which lines are executed. But it can also be used as a direct command to start a program at any given point. GOSUB 300, for example, acts like RUN 300. The difference is that it does not force a **CLR** operation and so leaves the variables intact.

Associated keyword: **ON**.

graphics   Any part of the display that is not recognisable as text, such as sprites, pictures, lines, circles, graphs.

graphics characters   Part of the character set, these are the non-alphanumeric characters which are displayed at the front of the keys. They can be used in the same way as any other characters in **strings** and **string variables**.

graphics tablet   A graphics aid which allows the user to create graphics on screen by drawing on a board. Some graphics tablets (known as digital tracers) work by reproducing the line traced by a moveable arm. More sophisticated tablets use light or pressure sensitive pads, and offer paintbrush as well as line drawing facilities.

hardware   The electronic and mechanical components of a computer in contrast to its software.

hexadecimal   A number system which uses 16 digits, 0123456789ABCDEF, representing the decimal numbers 10 to 15 by the

letters A to F. The word hexadecimal is often abbreviated to hex.

Like any other number system, the value of a digit depends on which column it is in. As hexadecimal is to the base 16 the column values increase in powers of sixteen. Thus the digit in the first column indicates the number of units, the second indicates the number of 16s, the third the number of 16×16s (256), the fourth the number of 16×16×16s (4096), and so on.

DECIMAL	HEX	BINARY
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

To convert a hex number to decimal, first find the decimal equivalent for any letter digit and then multiply by its column value, e.g.:

    D7 = 13×16 + 7×1 = 215
    F58C = 15×4096 + 5×256 + 8×16 +
    12×1 = 62860

A simple way of converting a number from decimal to hex is to break it down into 4096s, 256s, 16s, and units. Then change each number over 9 into its hex equivalent, e.g.:

    248 = 15×16 + 8 = F8
    50347 = 12×4096 + 4×256 + 10×16 +
    11×1 = C4AB

Because 16 is a power of two, hex serves as a kind of shorthand for binary. Machine code programmers find it more convenient to use than decimal. In hex, four bits can be represented by a single digit, and all one byte numbers can be shown as two hex digits. Most assemblers present numbers in hexadecimal form.

To distinguish a hex number from decimal, precede it with a $ sign.

high-level language A programming language that needs to be translated into **machine code** by an **interpreter** or a **compiler** before it can be run. A single statement in a high-level language such as **BASIC** generally represents a series of machine code in-

structions. Programs in high level languages
are therefore easier to write and understand
than programs written directly in machine
code. Usually they are also slower. Machine
code and **assembly language** are sometimes
referred to as low-level languages.

See **FORTH**; **LOGO**.

high-resolution graphics A display
mode which gives a screen made up of 320
horizontal dots by 200 vertical dots and allows
each dot (or pixel) to be turned on or off. In
high-resolution graphics the pixels are repre-
sented by bits in memory: each pixel is said to
be mapped on to a bit. The pixels and back-
ground can be assigned any one of 16 colours.
But, as in **character mode**, normally all the
pixels in each 8 by 8 group must take the same
colour. However, more than one colour can be
chosen if the multicolour option is selected.
For setting up a high-resolution screen, *see* **bit
map mode**.

IEEE (IEEE-488) The 'I-triple-E' is a stan-
dard **parallel** interface for connecting the
computer to another device, usually a disk
drive. Devices that run on this standard re-
quire an **IEEE** interface to be plugged into one

of the computer's **ports**.

IF A statement which allows the computer to make decisions. Used in conjunction with **THEN** it sets up a condition to be tested. IF the condition is true it executes the statement after the THEN statement. IF the condition is false the program passes on to the next line. (*See* **truth value**.) The following examples illustrate some of the ways in which it is used:

```
IF A>B THEN PRINT A;" IS ";GREATER
THAN ;"B
IF B$="" THEN B$=A$
IF N=3 THEN GOTO 200
IF W=3 AND X=0 THEN PRINT "I WIN"
```

There may be more than one statement after THEN, as in

```
100  IF A$="Z" THEN Y=Y+1:GOTO 300
200  GOTO 50
```

Note that if the condition is not true, i.e. A$ does not equal Z, then the program does not proceed to the second statement in line 100, but to line 200.

Associated keyword: **THEN**.

ILLEGAL DIRECT An **error message**: an attempt has been to give as a direct command an instruction that can only be used in a

program, e.g: **DEF FN**, **INPUT**.

ILLEGAL QUANTITY An **error message**: a number or variable is outside the computer's range. It is often caused by trying to POKE a value greater than 255.

immediate addressing Treats the **operand** as data rather than the address of data. Used with the **accumulator** and index registers, this mode operates on the byte following the **op code**. In assembly language the operand is always preceded by a # character to show that it is to be treated as immediate data, e.g. 'LDA #55' loads 55 into the accumulator, and 'CPY #120' compares the contents of the Y register with 120.

implied addressing Indicates that the instruction operates one of the registers. In this mode the **operand** (a register) is not specified since it is implied by the instruction itself. Implied addressing instructions occupy only one byte, e.g. **INY**, **CLC**, **RTS**.

INC A 6510 instruction mnemonic which INCreases the contents of a memory by one, and sets the zero flag to 1 if the result is 0, e.g.:
```
  INC $C108
```

BEQ LABEL

increases the contents of memory C108 by one. If its previous value was 255 the zero flag is set to Ø and causes a branch.

	Status register	N	V	B	D	I	Z	C
		√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
zero page	INC operand	E6	2	5
zero page, X	INC operand, X	F6	2	6
absolute	INC operand	EE	3	6
absolute, X	INC operand, X	FE	3	7

indexed addressing Adds the value of the Y or X **index registers** to a base address specified by the **operand**. This gives the effective address of the byte in memory that the instruction operates on, e.g. 'STA $C000,Y' stores the contents of the accumulator at location C000 + Y. If Y contains 5 then the address is C005. In **zero page** indexed addressing, the operand is only one byte long and the base address is between Ø and 255, e.g. 'LDA $6F,X'. This mode is often used to process a block of up to 256 consecutive bytes.

index registers The X and Y **registers**. Although their primary use is to provide an index for **addressing modes**, they can also

transfer data to and from the **accumulator** or a memory location. In addition they often serve as loop counters.

indirect addressing   Only used with the **JMP** instruction. The two bytes following the **op code** give the address of a memory location which in turn holds the address of the jump destination, e.g. 'JMP ($C450)' jumps to an address whose location is held at C450 and C451.

INPUT   A statement which allows the user to enter numbers or words into the computer while a program is running. INPUT reads characters from the keyboard into a variable. Unlike **GET**, it accepts more than one character and waits until the **RETURN key** has been pressed. To indicate that it is waiting for input the computer prints a question mark on the screen. The variable which follows the INPUT statement must be of the right type. For example, 'INPUT N' expects numbers. If you enter a letter the error message '?REDO FROM START' appears. It is also possible to follow INPUT with more than one variable. In response to '10 INPUT A,B,C' you can either type in three numbers separated by

commas and then press RETURN, or press RETURN after each. When the computer expects further input it prints '??'.

If text, in quotation marks and followed by a semi-colon, is inserted between INPUT and a variable, it prints a prompt message on the screen. When 'INPUT "ENTER YOUR NAME ";' is run, the screen displays 'ENTER YOUR NAME ?'.

Associated keywords: **GET**; **INPUT#**.

INPUT#   An input/output Statement. Using this command is the most common way of inputting data from a file on tape or disk. It reads up to 80 characters into a variable until it reaches a separator – either a return character (CHR$13), a comma, or a semi-colon. INPUT# must be followed by a file number previously given in an **OPEN** statement. Like INPUT it can take more than one variable, e.g.:

```
INPUT#1,A$
INPUT#2,A$,B$,N
```

Associated keywords: **GET#**; **CLOSE**; **PRINT#**; **OPEN**.

input/output (I/O)   The term covers any part of the computer's hardware or software

that is involved in communicating with an external device, e.g. an I/O **port** is one that can both input and output data.

INST/DEL key   Used to edit a program line or a **direct command**. Pressing the INST/DEL key deletes the character to the left of the **cursor** and closes up the line. When the **shift** key is held down at the same time INST/DEL inserts a space at the cursor position and shifts the following characters to the right.

See **screen editor**.

INT   An integer function which gives the integer part of an expression by stripping off the decimal fraction. For example, 'PRINT INT (2.56)' returns 2. If the number is negative it returns the next lower integer. One of its many uses is for finding the remainder of a number after division. This line will give the remainder when 18 is divided by 5:

    PRINT 18−(INT(18/5)×5)

INT does not supply the integer value of an expression to the nearest whole number. To do this add 0.5, as in:

    10 INPUT N
    20 PRINT INT(N+0.5)

integer variables   Store whole numbers between −32768 and +32767. The names of integer variables must end with a % sign, e.g. 'T2%', 'N%', 'FIRST%'. Operations involving integer variables are generally slower than those with **floating point variables**, but integer variables consume less memory. BASIC takes 2 bytes to store an integer variable, and 5 for a floating point variable. If a floating point number or variable is assigned to an integer variable, the fractional part of its value is stripped off, e.g.:

```
10  T% = 12.08
20  PRINT T%
```

prints 12.

interface   The hardware and software that allows two devices – usually the computer and an external device – to be connected. On the Commodore 64 the **ports** at the back of the machine provide connections to Commodore's own **disk drive** and **printer**, but devices from other manufacturers generally require extra interfaces to be plugged in. These take the form of cables or **cartridges** and accompanying software.

interpreter   The program which translates

BASIC programs into machine code. As **machine code** is the only language that the computer's **microprocessor** understands, high level languages like BASIC have to be converted before they can run. In contrast to a compiler, the interpreter turns a BASIC program into machine code while it is running by dealing with each statement in turn. If, for example, a program repeats a PRINT statement ten times in a loop, then that statement is interpreted ten times over. As the task of interpreting takes time, interpreted BASIC runs substantially slower than low level languages. The interpreter is itself a machine code program. In the Commodore 64 it is 8K long and is stored permanently in ROM at addresses 40960 to 49151.

interrupt When the computer's 6510 microprocessor receives an interrupt signal it stops executing the current program and jumps to a routine which handles whatever caused the interrupt. Then it resumes executing the program where it left off. Interrupts enable the 6510 to respond to external events or carry out regular tasks internally, while a program is running. They make it possible,

for example, to stop a program by pressing the **RUN/STOP key**: every 1/50th sec. the 6510 is interrupted by a signal from a timer and, among other things, scans the keyboard and checks for key presses.

There are three kinds of interrupts: **IRQ** and **NMI** hardware interrupts and a software interrupt caused by the **BRK** instruction. By intercepting the routine which handles IRQ interrupts machine code programmers can add their own interrupt-driven routines. These allow a task to be performed independently of another program.

See **raster**; **interrupt**; **wedge**.

INX A 6510 instruction mnemonic which INcreases the contents of the X **index register** by one, and sets the zero flag to 1 if the result is 0. Used with **indexed addressing** and to increase the value of X when it acts as a loop counter.

	Status register	N	V	B	D	I	Z	C
		$\swarrow$	–	–	–	–	$\swarrow$	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	INX	E8	1	2

INY A 6510 instruction mnemonic which

INcreases the contents of the Y **index register** by one and sets the zero flag to 1 if the result is 0. Used in the same way as **INX**.

Status register	N	V	B	D	I	Z	C
	✓	–	–	–	–	✓	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	INY	C8	1	2

IRQ A hardware **interrupt**. The 6510 microprocessor has two interrupt pins, IRQ and **NMI**. Unlike NMI interrupts, IRQ interrupts can be disabled by setting the interrupt flag in the **status register**. The instruction **SEI** disables interrupts; **CLI** enables them. When the 6510 receives an IRQ interrupt it stores the contents of the **program counter** and status register on the **stack**, and then jumps to an interrupt service routine via a vector in RAM. The IRQ vector is located at 788 and 789.

The IRQ service routine is called every 1/50th sec. Its main purpose is to scan the keyboard and update the interval timer which handles the TI and TI$ functions. By changing the address held in the IRQ **vector** a machine code program can be inserted into the normal routine. Such a program is said to be interrupt-

driven.
  See **time**; **wedge**.

JMP A 6510 instruction mnemonic which
JuMPs to a new location. JMP has two
addressing modes, absolute (direct) and in-
direct. In an absolute JMP the program bran-
ches to the address following the instruction.
In an indirect jump it branches to the address
held in the two bytes following the instruc-
tion, e.g. 'JMP ($C100)'. If locations C100 and
C101 hold the address 12288 then JMP bran-
ches to that address. Note that the 6510
microprocessor stores addresses in the order:
least significant byte followed by most signi-
ficant byte. In the example above, C100 con-
tains 0 and C101 contains 48. 12288 = 48 ×
256.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
absolute	JMP operand	4C	3	3
indirect	JMP (operand)	6C	3	5

joystick A moveable stick attached to a
base with a fire button. Joysticks are used
mainly as an alternative to the keyboard for

game control. There are two joystick ports at the right of the computer, for any joystick with a D-type plug. As well as the Commodore's own joysticks, Atari joysticks can be fitted.

The Commodore 64 takes digital as opposed to analogue joysticks. They contain four switches to register movement and one for the fire button. Moving the stick left, right, up, or down, closes a single switch; moving it diagonally closes two switches. The values of registers 56320 and 56321 indicate which switches are closed or open in joysticks fitted to ports 1 and 2 respectively. One of the first five bits is set to 0 when a switch is closed or the fire button is pressed, and set to 1 when a switch is open.

address:	bit 4	bit 3	bit 2	bit 1	bit 0	joystick:
53620	FIRE	RIGHT	LEFT	DOWN	UP	1
53621	FIRE	RIGHT	LEFT	DOWN	UP	2

bits 5, 6 and 7 not used

To check the movement of a joystick in port 1 use

```
D = 15 - (PEEK(56320) AND 15)
```

D gives the direction shown in the table opposite.

VALUE OF D	DIRECTION
0	—
1	UP
2	DOWN
3	—
4	LEFT
5	UP & LEFT
6	DOWN & LEFT
7	—
8	RIGHT
9	UP & RIGHT
10	DOWN & RIGHT

PEEK(56320) AND 16

returns 0 when the fire button is pressed. Substitute 56321 for 56320 to read port 2.

The following program shows how a joystick can control movement on screen:

```
10   SC=1564:CL=55836
20   D=15-(PEEK(56320)AND15)
30   M=0
40   IF D=1 THEN M=-40
50   IF D=2 THEN M=40
60   IF D=4 THEN M=-1
70   IF D=8 THEN M=1
80   IF SC+M < 1024 OR SC+M >
     2023 THEN 20
90   POKE SC,32
100  SC=SC+M:CL=CL+M
110  POKE SC,81:POKE CL,0
```

120 GOTO 20

JSR A 6510 instruction mnemonic which causes a Jump to a SubRoutine. JSR is equivalent to a **GOSUB** instruction in BASIC. When the computer executes JSR it stores the address of the next instruction on the **stack**. When the program meets an **RTS** instruction it pulls the return address off the stack and returns from the subroutine.

Status register	N	V	B	D	I	Z	C
	–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
absolute	JSR operand	20	3	6

Kernal The Commodore 64's **operating system**. It performs such tasks as reading the keyboard and printing what is typed on the screen, loading and saving programs, moving the cursor, and organising memory resources (*see* **bank switching**). The Kernal is an 8K **machine code** program held in ROM from addresses 57344 to 65535. It consists of a collection of subroutines, each one handling its own specific task. The address of each subroutine is given in a table known as a jump table located at the end of the ROM. When the

Kernal calls a routine it first consults the table
to find its address. CHROUT, for example is
the subroutine which prints a character to the
screen. Its address is held in the jump table at
64590. Many of these routines can be called *via*
the jump table by machine code programs in
RAM. Together with the BASIC **interpreter**
ROM, the Kernal uses the first 1K of RAM
(locations 0 to 1023) for storing its own vari-
ables.

   See **system variables**.

keyboard buffer   Whenever a key is press-
ed its keyboard code is stored in the keyboard
buffer, addresses from 631 to 640. The
keyboard buffer enables the user to enter char-
acters while the computer is occupied with
another task. Without it, characters might
sometimes get lost when typed rapidly. The
operating system extracts characters from the
buffer in the order they were stored. In prac-
tice, they are removed as soon as they are
stored. But while a program is running they
queue up until a **GET** statement is performed.
This means that GET occasionally picks up a
character from an earlier keypress. Location
197 holds the code of the current keystroke. It

can be used as an alternative to GET for reading the keyboard. The following program PEEKs location 197 and prints the code associated with each keystroke. If no key is pressed, the value 64 is returned.

```
10  K=PEEK(197)
20  PRINT K;" ";
30  GOTO 10
```

LAN   *See* **network**.

LDA   A 6510 instruction mnemonic which LoaDs the **Accumulator** with a given value or the contents of a memory location. If 0 is loaded the zero flag is set to 1. This instruction is probably used more often than any other. It has 8 **addressing modes**, of which here are 3:

Status register	N	V	B	D	I	Z	C
	√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	LDA # operand	A9	2	2
zero page	LDA operand	A5	2	3
zero page, X	LDA operand, X	B5	2	4
absolute	LDA operand	AD	3	4
absolute, X	LDA operand, X	BD	3	4*
absolute, Y	LDA operand, Y	B9	3	4*
(indirect, X)	LDA (operand, X)	A1	2	6
(indirect), Y	LDA (operand), Y	B1	2	5*

*Add 1 if page boundary is crossed.

'LDA #55' loads 55 into the accumulator. 'LDA $FB' loads the byte at FB into the accumulator. 'LDA (252),Y' loads the byte from the address held at location 252 + Y.

LDA is typically used to transfer data to and from memory, either for storage or for arithmetic and logical operations.

LDX A 6510 instruction mnemonic which LoaDs a given number or the contents of a memory location into the X **index register**. It acts in the same way as **LDA** but offers fewer **addressing modes**.

	Status register	N	V	B	D	I	Z	C
		√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	LDX * operand	A2	2	2
zero page	LDX operand	A6	2	3
zero page, Y	LDX operand,Y	B6	2	4
absolute	LDX operand	AE	3	4
absolute, Y	LDX operand,Y	BE	3	4*

*Add 1 if page boundary is crossed.

LDY A 6510 instruction mnemonic which LoaDs a given number or the contents of a memory location into the Y **index register**. It acts in the same way as **LDA** but offers fewer **addressing modes**.

		Status register	N	V	B	D	I	Z	C
			√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	LDY # operand	A0	2	2
zero page	LDY operand	A4	2	3
zero page, X	LDY operand, X	B4	2	4
absolute	LDY operand	AC	3	4
absolute, X	LDY operand, X	BC	3	4*

* Add 1 if page boundary is crossed.

least significant bit Bit 0, which can only be worth 1, is the least significant. Bit 7, the most significant, is also used in some situations (*see* **two's complement**) as a flag to indicate negative numbers.

least significant byte Where numbers greater than 255 need to be POKEd into memory, two, or occasionally three, bytes are used. The number is then stored in the form:

number = 1st byte + 256 * 2nd byte (+ 256 * 256 * 3rd byte)

The first byte is the least significant byte, and the final byte is the most significant.

LEFT$ A string function used to extract one or more characters from a string, starting at the left-hand end of the string. It has the form:

LEFT$(A$,N)

where A$ is the source string and N specifies the number of characters required, e.g. 'LEFT$("DICTIONARY",7)' extracts the substring DICTION. If N is zero the function returns an empty string.

In the following program it is used with **MID$** to separate a first name from a surname:

```
10  INPUT "YOUR FULL NAME";A$
20  FOR N=1 TO LEN(A$)
30  IF MID$(A$,N,1)=" " THEN S=N
40  NEXT
50  PRINT "HELLO ";LEFT$(A$,S)
```

Associated keywords: **LEN**; **MID$**; **RIGHT$**.

LEN A string function which counts the number of characters in a string. For example, there are nine letters and one space in the name JOHN SMITH, so 'L=LEN("JOHN SMITH")' assigns the value 10 to L.

In the following program, LEN is used with MID$ to reverse the letters in a word:

```
10  INPUT "TYPE IN A WORD";A$
20  FOR N=1 TO LEN(A$)
30  B$=MID$(A$,N,1)+B$
40  NEXT
50  PRINT B$
```

Associated keywords: **LEFT$**; **MID$**; **RIGHT$**.

LET A statement which assigns a value to a variable, as in

```
LET A$="TABLE"
LET B%=22
LET N=3.65
```

However 'LET A$="TABLE"' is the same as 'A$="TABLE"'. The word LET is optional, and is often omitted.

light pen A device in the shape of a pen which allows the user to create pictures in **high resolution graphics** by moving the tip around the front of the screen. It can also be used to select items from a **menu** by pointing to screen boxes. A light pen uses a photosensitive cell to detect the TV's raster beam. By sending a signal when the beam passes beneath it, it enables the **VIC** chip to work out its position. The VIC registers 19 and 20 give the X and Y coordinates of the light pen's position.

Before a light pen can "draw" on the screen, it requires a program to read its position and plot the corresponding pixels.

line number Every program line must be preceded by a line number, which can be from 0 to 63999. Lines are deleted by typing the line number and pressing RETURN. It is a good idea to number lines in steps of 10 so that new lines can be inserted later at the numbers in between.

LIST A command which makes the computer print out a program on the screen, line by line. If the program is a long one it will **scroll** down the screen too quickly to be read. It can be slowed down by pressing the **CTRL key**. To stop it press the **RUN/STOP key**.

LIST followed by a line number prints a single line. Or you can list a range of lines:

    LIST 120–200

displays all the lines from 120 to 200.

    LIST –120

lists a program from the beginning up to line 120. And

    LIST 120–

lists the lines from 120 to the end.

When it is preceded by printer commands, LIST prints a program out on a printer, e.g.: 'OPEN4,4:CMD4:LIST'. A printout of a program is generally known as a listing.

LIST is usually entered as a direct command but can be used in a program.

LOAD (1) A command which transfers a program from tape or disk into memory. On its own, LOAD will load in the first program it finds on tape. It can also be followed by three optional parameters: LOAD "file-name", device, address.

When the file-name is given, LOAD will look for a particular program: 'LOAD "GAME"' searches the tape until it finds the program GAME. If you are loading from disk the file-name must be specified along with the device number. 8 is the device number for disk, 1 for tape. To load the program '"PROG3"' from disk you would enter: 'LOAD "PROG3",8'. There is generally no need to supply a device number for tape loading. If none is given it is assumed to be 1. Normally, programs load into the BASIC program area starting at address 2048. If the last parameter is 1 then a program will be loaded at the memory location from which it was SAVEd. This option can be used for loading machine code programs or blocks of data. (*See* **machine code**.) To load data files

see **cassette files, relative files, sequential files**. When the LOAD is executed as a direct command it forces a **CLR** statement to be performed. When it is used within a program it loads and **RUN**s another program but leaves the variables intact. Note that the second program will overwrite the first.

Associated keywords: **SAVE**; **VERIFY**.

LOAD examples	
LOAD	loads next program on tape
LOAD "PROGNAME"	searches for then loads PROGNAME
LOAD "PROGNAME",1,1	loads program into memory at the location at which it saved from
LOAD A$	loads program whose name is held in A$
LOAD "PROGNAME",8	loads PROGNAME from disk
LOAD "*",8	loads first program found on disk
LOAD "PROGNAME",8,1	loads first program from disk at the location at which it was saved from
LOAD "$",8	loads disk directory

(2) An **error message**: there is a problem with the program on tape, e.g. the program has been corrupted.

local area network (LAN) *See* network.

LOG A floating-point function which calculates the natural logarithm of a number to the base e. In common with other mathematical

functions on the Commodore 64, it gives the result to an accuracy of nine decimal places. To convert a natural logarithm to a common logarithm to the base 10, divide it by LOG(10). For example LOG(5) gives 1.60943791 while LOG(5)/LOG(10) gives 0.698970004, the logarithm of 5 to the base 10.

Associated keyword: **EXP**.

logical operators AND, OR, NOT, which can be used with **relational** and **arithmetic operators**, together with **strings**, numbers, and **variables**, to form expressions which can have a value of 'true' or 'false'. The logical operators determine the truth value of an expression depending on which conditions are met, e.g. the expression 'Y>9 AND X=0' is evaluated as true only if 'Y' is greater than '9' at the same time as 'X' equals '0'.

Logical operators also act as bitwise operators, comparing the bits of one number with the bits of another, e.g. 'PRINT 18 OR 137' gives 155.

See **truth table**.

LOGO A high level language originally intended for educational use. Although some versions of LOGO are as extensive as BASIC,

it is primarily used to create turtle graphics. A
"turtle" is a small robot which holds a pen and
draws as it moves. LOGO instructions give
the turtle a path to follow, in course of which
it draws pictures or patterns on the paper it is
placed on. One of the attractions of the lan-
guage for children is that the instructions, such
as FORWARD, PENUP, PENDOWN,
RIGHT and LEFT, are familiar and easy to
learn.

   LOGO is more commonly used to create
graphics on screen, where it represents the
turtle by a small triangle. Like FORTH it has
the merit of allowing the user to define new
instructions. For example, the following com-
mands draw a triangle:

```
FORWARD 50
LEFT 120
FORWARD 50
LEFT 120
FORWARD 50
LEFT 120
```

By giving it a name, such as TRIANGLE, this
sequence of commands, can be defined as a
single instruction.

   LOGO can be loaded in from cassette, disk,
or cartridge and run as a BASIC alternative.

LSR A 6510 instruction mnemonic which moves a byte in the **accumulator** or a memory location one bit to the right. Bit 7 becomes 0 and bit 0 moves into the carry flag. LSR has the effect of dividing the value of a byte by two, e.g.:

```
    LDA #32
    LSR
    LSR
    LSR
```

leaves 4 in the accumulator.

Status register		N	V	B	D	I	Z	C
		0	–	–	–	–	↓	↓

addressing mode	assembly language form	op code	No. bytes	No. cycles
accumulator	LSR A	4A	1	2
zero page	LSR operand	46	2	5
zero page, X	LSR operand, X	56	2	6
absolute	LSR operand	4E	3	6
absolute, X	LSR operand, X	5E	3	7

machine code The language understood by the computer's **microprocessor**, the 6510. Programs written in any other language need to be translated into machine code before they can be executed.

See **interpreter**; **compiler**.

Not only do machine code programs run

many times faster than BASIC but they also allow the programmer to access parts of the computer that are closed to BASIC. Making use of the interrupts, for example, is only possible in machine code. Almost all commercial software is written in machine code, as is the computer's operating system and BASIC Interpreter.

The 6510 instruction set contains 56 machine code instructions. They are usually referred to by their assembly language mnemonics, but are stored in memory and executed as **op codes** – 1-byte numbers in the range 0 to 255. An instruction may take a number of different forms depending on its addressing mode.

ADC	add memory to accumulator with carry
AND	AND memory with accumulator
ASL	shift left one bit (memory or accumulator)
BCC	branch on carry clear
BCS	branch on carry set
BEQ	branch on result zero
BIT	test bits in memory with accumulator
BMI	branch on result minus
BNE	branch on result not zero
BPL	branch on result plus
BRK	force break
BVC	branch on overflow clear
BVS	branch on overflow set
CLC	clear carry flag
CLD	clear decimal mode

CLI	clear interrupt disable bit
CLV	clear overflow flag
CMP	compare memory and accumulator
CPX	compare memory and index X
CPY	compare memory and index Y
DEC	decrement memory by one
DEX	decrement index X by one
DEY	decrement index Y by one
EOR	exclusive-OR memory with accumulator
INC	increment memory by one
INX	increment index X by one
INY	increment index Y by one
JMP	jump to new location
JSR	jump to new location saving return address
LDA	load accumulator with memory
LDX	load index X with memory
LDY	load index Y with memory
LSR	shift right one bit (memory or accumulator)
NOP	no operation
ORA	OR memory with accumulator
PHA	push accumulator on stack
PHP	push processor status on stack
PLA	pull accumulator from stack
PLP	pull processor status from stack
ROL	rotate one bit left (memory or accumulator)
ROR	rotate one bit right (memory or accumulator)
RTI	return from interrupt
RTS	return from subroutine
SBC	subtract memory from accumulator with borrow
SEC	set carry flag
SED	set decimal mode
SEI	set interrupt disable status
STA	store accumulator in memory
STX	store index X in memory
STY	store index Y in memory
TAX	transfer accumulator to index X
TAY	transfer accumulator to index Y
TSX	transfer stack pointer to index X

TXA	transfer index X to accumulator
TXS	transfer index X to stack pointer
TYA	transfer index Y to accumulator

Each form has a different op code. Thus the instruction JMP can take two forms depending on whether it jumps to a location directly or indirectly. They are represented in assembly language as

```
JMP  operand
JMP  (operand)
```

and their op codes in hexadecimal are 4C and 6C.

Although there are many ways of classifying the instructions, most of them fall into the following broad categories:

DATA TRANSFER INSTRUCTIONS. Move data between registers and memory, e.g. **LDA**, **STY**.

REGISTER TRANSFER INSTRUCTIONS. Move data between registers, e.g. **TXA**, **TSX**.

CONDITIONAL BRANCH INSTRUCTIONS. Branch to a different part of the program when a flag is set, e.g. **BPL**, **BNE**.

JUMP AND PROGRAM CONTROL INSTRUCTIONS. Equivalent to the BASIC commands GOTO and GOSUB, e.g. **JMP**, **JSR**.

INCREMENT/DECREMENT INSTRUCTIONS. Alter

the value of registers or memory by one, e.g. **INC**, **DEY**.

ARITHMETIC AND LOGICAL INSTRUCTIONS. Perform operations on the contents of the accumulator or memory, e.g. **ADC**, **ORA**.

STACK TRANSFER INSTRUCTIONS. Transfer the contents of the accumulator or status register to and from the stack, e.g. **PHA**, **PHP**.

COMPARE INSTRUCTIONS. Test the contents of a memory location with the contents of the accumulator or index registers, e.g. **CMP**, **CPY**.

SHIFT AND ROTATE INSTRUCTIONS. Move each bit in the accumulator or memory to an adjacent position, e.g. **LSR**, **ROL**.

FLAG INSTRUCTION. Alter the flags in the status register, e.g. **CLC**, **SEI**.

Information on each instruction is given in its dictionary entry in a table with the following columns:

ADDRESSING MODE. The way it operates on data or addresses.

ASSEMBLY LANGUAGE FORM. The instruction itself followed by its **operand** (if any).

OP CODE. The single byte by which an instruction is stored in memory, given in **hexadecimal**.

NUMBER OF BYTES. The number of bytes occupied by an instruction and its operand.

NUMBER OF CYCLES. The number of clock cycles taken to execute an instruction. If the instruction crosses a page boundary it takes an extra cycle to execute. This is indicated by an asterisk.

See **zero page**.

Each dictionary entry also shows which flags in the status register may be affected when an instruction is executed.

See **register**.

A machine code program can be stored anywhere in **RAM**. The area from 49152 to 53247 is particularly suitable since it cannot be overwritten by a BASIC program.

To enter a machine code program it is easiest to use an **assembler**. The alternative is to hand assemble a program and store it in memory with a hexloader. Hand assembly means translating each instruction into its hexadecimal **op code**. A hexloader takes the instructions from DATA statements, converts them into decimal and then POKEs them into memory.

machine code monitor   A program for

entering and testing machine code. Monitors allow the programmer to examine and alter sections of **RAM** or the **registers**, and move blocks of code. They may also provide a facility for stepping through a machine code program one instruction at a time. Some monitors include an **assembler** and **dis-assembler**.

mask   Used to read or alter one or more **bits** in a **byte**. Many of the computer's facilities such as sprites and sound are only available by setting (or examining) particular bits in a register to 1 or 0, while leaving the rest unchanged. Masks (sometimes called bit masks) employ the logical operators AND and OR. AND allows bits to be read or set to zero. If a bit in the number which acts as a mask is 0 then the corresponding bit in the number being read is ignored; while if a bit in the mask is 1 the value of its corresponding bit is returned. Thus ANDing a number with 15 gives the value of its bottom four bits since 15 in binary is 00001111. For example:

```
      181   10110101
AND    15   00001111
       ─────────────
        5   00000101
```

To read bit number N in byte B use
```
PEEK(B) AND 2↑N
```
To set bit N to zero use
```
POKE B, PEEK(B) AND (255−2↑N)
```
OR allows particular bits to be set to one. When a bit in the mask is one then the corresponding bit is set to one, whether it is zero or one already. For example, ORing a number with 136 sets bits 3 and 7 to one but leaves the other bits unchanged:

```
     113  01110001
OR 136  10001000
     249  11111001
```

Use this formula to set bit N in byte B to one:
```
POKE B, PEEK(B) OR 2↑N
```

memory   The part of the computer's hardware that stores data of any kind. Numbers, characters, variables, programs, etc., are all held in memory.

See **ROM**; **RAM**; **address**; **bit**; **byte**.

memory map   Shows how the computer's 64K of memory is allocated to different parts of the system – **programs**, the BASIC **interpreter**, **screen memory**, and so on, e.g. colour memory extends from 55296 to 56319.

Note that some addresses are occupied by either **ROM** or **RAM**. The BASIC interpreter and operating system are normally in place from 40960 to 49151 and 57344 to 65535, but can be switched out to give a different memory configuration. Similarly, the program area extends to 40959 unless a cartridge is plugged in; while the character generator ROM is continuously switched in and out.

See **bank switching**.

addresses	contents	
65535	8K operating system ROM or RAM	
57344		
56320	I/O RAM	4K character generator ROM
55296	1K colour memory	
53248	VIC and SID registers	
49152	4K RAM	
40960	8K BASIC interpreter ROM or RAM	
32768	BASIC program area or 8K cartridge ROM	
2048	BASIC program area	
1024	1K screen memory	
0	system variables	

memory map appendix

ADDRESS	DESCRIPTION
0	6510 Data direction register
1	6510 I/O register
2	not used
3–4	vector for floating point-integer conversion
5–6	vector for integer-floating point conversion
7	BASIC counter; search for end of statement
8	scan for quotes at end of string flag
9	cursor position on line after TAB
10	load/verify flag
11	BASIC input buffer pointer/number of subscripts
12	default DIM flag
13	BASIC variable flag: $FF=string, $00=numeric
14	numeric variable flag: $80=integer, $00=numeric
15	DATA scan/LIST quote/memory flags
16	subscript/FNx flags
17	INPUT/GET/READ flag
18	ATN/comparison result flags
19	INPUT prompt flag
20–21	BASIC temporary store for integers
22	pointer to temporary string stack
23–24	last temporary string vector
25–33	temporary string stack
34–37	utility pointer area
38–42	holds product of multiply
43–44	start of BASIC pointer
45–46	start of BASIC variables/end of program pointer
47–48	start of arrays pointer
49–50	start of arrays/end of variables pointer
51–52	start of strings pointer
53–54	end of strings pointer
55–56	top of program area pointer
57–58	current BASIC line number
59–60	previous BASIC line number
61–62	pointer to statement for CONT
63–64	current DATA line number
65–66	current DATA item pointer

ADDRESS	DESCRIPTION
67–68	INPUT routine vector
69–70	current variable name
71–72	current variable pointer
73–74	FOR-NEXT variable pointer
75–96	miscellaneous pointers/work area
97–112	floating point accumulator workspace
113–114	cassette buffer pointer
115–138	CHRGET subroutine: get next BASIC character
139–143	RND function seed value
144	status word ST
145	STOP/REVERSE key flags
146	timing constant for tape
147	LOAD/VERIFY flag
148	serial buffered character flag
149	serial buffered character
150	cassette sync number
151	register save
152	number of OPEN files
153	input device number
154	output device number
155	tape character parity
156	tape byte received flag
157	direct/program mode flag
158	tape pass 1 error log
159	tape pass 2 error log
160–162	internal timer used by TI/TI$
163–164	temporary data area
165–182	cassette/RS232 data area
183	length of filename
184	logical file number
185	secondary address
186	device number
187–188	file name pointer
189–192	cassette/RS232 data
193–194	I/O start address
195–196	Kernal setup pointer
197	last key pressed

ADDRESS	DESCRIPTION
198	number of characters in keyboard buffer
199	reverse characters flag
200	end of line for INPUT pointer
201–202	cursor position at start of INPUT
203	current key pressed
204	cursor blink flag: 0 blink on; 1 blink off
205	cursor blink delay
206	character under cursor
207	cursor on/off flag
208	INPUT/GET flag
209–210	cursor row position pointer
211	cursor column position pointer
212	cursor inside quotes flag
213	40/80 screen line length
214	current line number of cursor position
215	ASCII value of character printed
216	number if INS outstanding
217–242	screen line link table
243–244	current location in colour memory pointer
245–246	keyboard decode table pointer
247–248	RS232 input buffer pointer
249–250	RS232 output buffer pointer
251–254	free zero page locations for user's program
255	BASIC temporary data area
256–266	floating point – ASCII work area
256–318	tape error log
256–511	6510 stack
512–600	BASIC input buffer
601–610	logical file table
611–620	device number table
621–630	secondary address table
631–640	keyboard buffer
641–642	operating system's start of RAM pointer
643–644	operating system's end of RAM pointer
645	serial timeout flag
646	colour code of current character
647	colour under cursor

ADDRESS	DESCRIPTION
648	screen memory high byte
649	maximum size of keyboard buffer
650	key autorepeat flag: 0=cursor, 128=all
651	delay time before key repeat
652	delay time between key repeat
653	SHIFT/CTRL/C= key press flag
654	last SHIFT/CTRL/C= press flag
655–656	keyboard table setup pointer
657	SHIFT mode enable/disable
658	auto scroll down flag: 0=on
659–670	RS232 data
671–672	IRQ vector during tape I/O
673	NMI interrupt control
674	timer A control log
675	interrupt log
676	timer A enable flag
677	screen row marker
678	PAL/NTSC flag
679–767	not used
768–769	vector for error message routine
770–771	vector for BASIC warm start
772–773	vector for convert to token
774–775	vector for convert token to ASCII
776–777	vector for start new BASIC code
778–779	vector for perform arithmetic function
780	storage for A register during SYS
781	storage for X register during SYS
782	storage for Y register during SYS
783	storage for status register during SYS
784–786	USR JMP followed by address
787	not used
788–789	IRQ vector
790–791	BRK vector
792–793	NMI vector
794–795	OPEN vector
796–797	CLOSE vector
798–799	set input device vector

ADDRESS	DESCRIPTION
800–801	set output device vector
802–803	restore I/O vector
804–805	input vector
806–807	output vector
808–809	test STOP key vector
810–811	GET vector
812–813	abort I/O vector
814–815	user defined vector
806–817	LOAD vector
818–819	SAVE vector
820–827	not used
828–1019	cassette buffer
1020–1023	not used
1024–2023	screen memory
2040–2047	sprite pointers
2048–40959	program area
32768–40959	ROM cartridge area
40960–49151	8K BASIC interpreter ROM
49152–53247	4K RAM for data storage or machine code
53248–57343	4K character generator ROM
53248–53294	VIC registers
54272–54300	SID registers
55296–56319	colour memory
56320–56591	I/O RAM
57344–65535	8K operating system ROM

menu A selection screen where the program options are displayed, with a routine to direct the program to the user's choice. Control will normally return to this screen at the end of each section.

merge The facility for joining two programs by loading one program from cassette or

disk while the other is already in **RAM**. There are a number of ways of merging programs although there is no BASIC command to do so. The simplest way is as follows:

1: Note the program start address by PEEKing the pointers at 43 and 44. 'PRINT PEEK(43), PEEK(44)'. This normally gives 1 and 8.

2: Alter their contents to point to the end of the program, 'POKE 43, PEEK(45) −2: POKE 44, PEEK(46)'.

3: Load another program.

4: Restore the original start address, 'POKE 43, 1:POKE 44,8'.

This procedure only works if the second program has higher **line numbers** than the first.

microprocessor The single chip that executes programs and in doing so carries out all data processing in the computer. Sometimes referred to as the central processing unit, the microprocessor receives and stores data in memory, and performs operations on data. The Commodore 64 uses an 8-bit 6510 microprocessor, which can only transfer and operate on 8 bits of data at a time. However it has 16 address lines which allow it to move

data – 8 bits at a time – to and from 2^{16} (65536) different memory locations. In other words, it can address 64K of memory. To store and operate on data the 6510 has 6 internal **registers**, 5 of them 8-bits wide and one 16-bit register which holds addresses. Each group of 8 bits represents a **binary** number between 0 and 255. Depending on the order in which it receives them the 6510 treats certain numbers as instructions, others as data. In all there are 56 different types of instruction, and together they make up the **machine code** instruction set.

   In almost all respects the 6510 is identical to the widely-used 6502 microprocessor, and shares the same instruction set. The major difference between the two is that the 6510 has an internal 8-bit input/output port which it uses for **bank switching** and to control the cassette unit. Along with the Z80, the 6502 is the most widely used microprocessor in home computers.

MID\$ A string function which returns one or more characters from within a given character string. This function is more flexible than LEFT\$ or RIGHT\$ as it can extract characters

from any point in the source string. It takes the form MID$(A$,S,X) which gives a substring X characters from the string A$ starting at position S, e.g.:

```
10  A$="IMMEDIATELY"
20  PRINT MID$(A$,3,5)
```

prints MEDIA.

If the last value 'X' is left out, the function assumes that the rest of the string is wanted, from the start position to the end. Thus 'MID$ ("WORDPROCESSOR",5)' gives 'PRO-CESSOR'.

Associated keywords: **LEFT$**; **RIGHT$**.

modem   A device which allows computers to communicate with each other over the telephone lines. By means of a modem, a user can link up with other owners *via* bulletin boards, or access viewdata systems such as Prestel, which hold large amounts of information on mainframe computers.

monitor   An alternative display unit to a television. Because monitors take a **composite video** signal from the **audio/video** port they give a clearer and steadier picture than televisions. **Wordprocessors** which offer an 80-column display need to be used with a

monochrome monitor.

most significant   *see* **least significant.**

multicolour bit map mode A **high-resolution** mode in which each **pixel** can take one of four different colours. In the standard **bit map mode** all the pixels in a character space take the same colour. The multicolour option gives a greater choice of colour at the cost of reduced pixel resolution: colours can only be assigned to pairs of pixels. In effect this halves the resolution from 320 by 200 to 160 by 200.

BIT PAIR	COLOUR INFORMATION
00	background 53281
01	top 4 bits of screen memory, usually 1024–2047
10	bottom 4 bits of screen memory 1024–2047
11	colour memory 55296–56295

In this mode each pixel is represented by a pair of bits in the bit map area. Depending on the value of its bit pair, a pixel's colour is determined by the colour code in one of four locations: the top and bottom four bits of the corresponding byte in screen memory; the corresponding byte in colour memory; and the

background colour register. Note that in bit map mode the screen memory, which extends from 1024 to 2047, stores colour information rather than character codes. To select bit map mode set bit 5 in location 53265, as follows:

```
POKE 53265,PEEK(53265) OR 32
```

The multicolour option can then be selected by setting bit 4 in location 53270:

```
POKE 53270,PEEK(53270) OR 16
```

To turn both options off enter:

```
POKE 53265, PEEK(53265) AND 223
POKE 53270,PEEK(53270) AND 239
```

multicolour mode A display mode which allows characters to take up to 4 colours. High resolution graphics and sprites can also be multicoloured. *See* **bit map**; **multicolour sprites**.

Multicolour mode is selected by setting bit 4 of location 53270 to 1. If the display is already in **character mode** then any character with a colour code of 8 or more becomes multicoloured. In other words, each multicoloured character must have bit 3 of its corresponding byte in **colour memory** set to 1. Characters whose colour code is less than 8 are displayed in the normal way. The following program

demonstrates multicolour mode by printing
two rows of the alphabet. When a key is
pressed multicolour mode is selected for the
second row (in line 60). Pressing a key again
turns it off (line 80).

```
10   PRINT CHR$(147)
20   B=0:GOSUB 100
30   PRINT CHR$(13);
40   B=8:GOSUB 100
50   GET A$:IF A$="" THEN 50
60   POKE 53270,PEEK(53270) OR 16
70   GET A$:IF A$="" THEN 70
80   POKE 53270,PEEK(53270) AND 239
90   GOTO 50
100  FOR N=65 TO 91
110  POKE 646,RND(0)*8+B
120  PRINT CHR$(N);
130  NEXT
140  RETURN
```

The normal character set is generally not
recognisable in multicolour mode. But **user
defined characters** can be designed specifi-
cally for this mode. The reasons for this lie in
the way the colours are assigned. Normally
the dots or pixels in a character space can take
only one colour. But each pixel in the 8 by 8
block can be on or off. In multicoloured mode

colours are assigned to horizontal pairs of pixels. This has the effect of halving the resolution: 1 pixel now has the width of 2 standard pixels.

It now takes 2 bits to represent a pixel in memory, and each pair of bits determines what colour the pixel takes. The colours are assigned as follows:

BIT PAIR	COLOUR	ADDRESS
00	background	53281
01	multicolour 1	53282
10	multicolour 2	53283
11	foreground	colour RAM (lower 3 bits)

To set colours 1 and 2 POKE the required colour codes into locations 53282 and 53283. Colour 0 is the screen background colour, and colour 4 is set by the code in the corresponding colour memory byte. As only the lower 3 bits at each colour memory location determine multicolour 4, it can only be assigned codes 0–7. The fourth bit, however, must be set to 1, so 8 needs to be added to the normal colour code, e.g. 'POKE colour memory location, 15' makes multicolour 3 yellow at a given location.

multicolour sprites Like multicoloured

characters, **sprites** can be given up to 4 colours, although one colour is that of the screen background. The colours are determined by pairs of bits in the sprite definitions. This means that the horizontal resolution is cut by half: each pair of pixels must take the same colour.
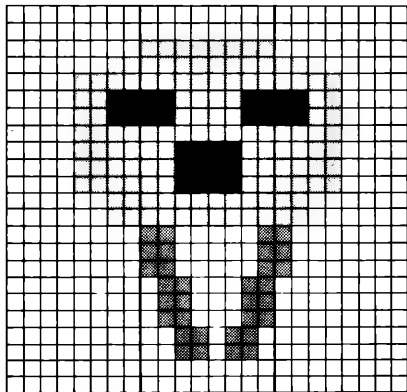
background	multicolour 1	sprite colour	multicolour 2
☐ = 00	▧ = 01	▨ = 10	■ = 11

The colours associated with each bit pair are given below. Bit pair 00 takes the background screen colour, and so disappears. Multicolours

1 and 2 are the same for all multicoloured sprites. The sprite colour is set in the normal sprite colour **registers**, 53287 to 53294 (V+39 to V+46).

BIT PAIR	COLOUR	REGISTER
00	background colour	53281
01	multicolour 1	53285
10	sprite colour	53287–53294
11	multicolour 2	53286

To turn on a multicolour sprite set the corresponding bit in register 53276 (V+28) to 1 by using the statement 'POKE 53276,PEEK(53276) OR (2↑SN)' where 'SN' is the sprite number.
POKE 53276,PEEK(53276) AND (255-2↑SN)

multi-statement line A program line with more than one statement. Each statement must be separated by a colon. As the computer takes 2 bytes to store a line number, reducing the number of lines by using multi-statements saves memory space. But note that REM should not appear as the first statement and IF . . . THEN passes control to the next line when a condition is false.

music There are a number of ways of playing music on the Commodore 64. At the most

advanced level, sheet music can be translated and played in three voices simultaneously and in a variety of different instrument sounds. To synthesise an instrument exactly and coordinate the timing of multiple voices generally involves using the **SID** chip's more specialised facilities – synchronisation, resonance, filter, and ring modulation, but satisfactory results can also be achieved with one voice by setting only the **envelope** and **waveform** parameters. The simplest method of playing a tune in one voice is to store each note's frequency and duration in **DATA** statements. This is the method used in the following program which plays the first eight bars of *Greensleeves*:

```
10  SD=54272: TE=40
20  DIM N(2,20)
30  FOR T=1 TO 19
40  READ F,D
50  FH=INT(F/256):FL=F-256×FH
60  N(0,T)=FL:N(1,T)=FH:N(2,T)=D×TE
70  NEXT
80  FOR T=SD TO SD+24
90  POKE SD,0
100 NEXT
110 POKE SD+24,15
120 POKE SD+5,9
```

```
130   POKE SD+6,0
140   FOR T=1 TO 19
150   POKE SD,N(0,T):POKE SD+1,N(1,T)
160   POKE SD+4,33
170   FOR D=1 TO N(2,T):NEXT
180   POKE SD+4,32
190   NEXT
200   DATA 5407,4,6430,8,7217,4,8101,8
210   DATA 9094,2,8101,2,7217,8,6069,4
220   DATA 4817,8,5407,2,6069,2,6430,8
230   DATA 5407,4,5407,6,5103,2,5407,4
240   DATA 6069,8,5103,4,4050,8
```

By changing the value of 'TE' in line 10 the tempo at which the tune is played can be speeded up or slowed down.

INSTRUMENT	ATT/DEC	SUS/REL	WAVEFORM	PULSE WIDTH
trumpet	96	128	sawtooth	—
violin	168	169	sawtooth	—
piano	9	9	pulse	1000
flute	154	0	triangle	—
harpsichord	9	0	sawtooth	—
accordeon	102	240	triangle	—
organ	0	242	sawtooth	—
clarinet	101	197	pulse	2048

Instrument sound can be changed by altering the envelope settings in lines 120 and 130 together with the waveform setting in line 160. The above table provides some possible

settings to approximate the sound of different instruments.

music note values

octave	decimal	frequency high	low
C–0	268	1	12
C#–0	284	1	28
D–0	301	1	45
D#–0	318	1	62
E–0	337	1	81
F–0	358	1	102
F#–0	379	1	123
G–0	401	1	145
G#–0	425	1	169
A–0	451	1	195
A#–0	477	1	221
B–0	506	1	250
C–1	536	2	24
C#–1	568	2	56
D–1	602	2	90
D#–1	637	2	125
E–1	675	2	163
F–1	716	2	204
F#–1	758	2	246
G#–1	803	3	35
G#–1	851	3	83
A–1	902	3	134
A#–1	955	3	187
B–1	1012	3	244
C–2	1072	4	48
C#–2	1136	4	112
D–2	1204	4	180
D#–2	1275	4	251
E–2	1351	5	71
F–2	1432	5	152
F#–2	1517	5	237

octave	decimals	frequency	
		high	low
G–2	1607	6	71
G#–2	1703	6	167
A–2	1804	7	12
A#–2	1911	7	119
B–2	2025	7	233
C–3	2145	8	97
C#–3	2273	8	225
D–3	2408	9	104
D#–3	2551	9	247
E–3	2703	10	143
F–3	2864	11	48
F#–3	3034	11	218
G–3	3215	12	143
G#–3	3406	13	78
A–3	3608	14	24
A#–3	3823	14	239
B–3	4050	15	210
C–4	4291	16	195
C#–4	4547	17	195
D–4	4817	18	209
D#–4	5103	19	239
E–4	5407	21	31
F–4	5728	22	96
F#–4	6069	23	181
G–4	6430	25	30
G#–4	6812	26	156
A–4	7217	28	49
A#–4	7647	29	223
B–4	8101	31	165
C–5	8583	33	135
C#–5	9094	35	134
D–5	9634	37	162
D#–5	10207	39	223
E–5	10814	42	62
F–5	11457	44	193
F#–5	12139	47	107

octave	decimals	frequency	
		high	low
G–5	12860	50	60
G#–5	13625	53	57
A–5	14435	56	99
A#–5	15294	59	190
B–5	16203	63	75
C–6	17167	67	15
C#–6	18188	71	12
D–6	19269	75	69
D#–6	20415	79	191
E–6	21629	84	125
F–6	22915	89	131
F#–6	24278	94	214
G–6	25721	100	121
G#–6	27251	106	115
A–6	28871	112	199
A#–6	30588	119	124
B–6	32407	126	151
C–7	34334	134	30
C#–7	36376	142	24
D–7	38539	150	139
D#–7	40830	159	126
E–7	43258	168	250
F–7	45830	179	6
F#–7	48556	189	172
G–7	51443	200	243
G#–7	54502	212	230
A–7	57743	225	143
A–7	61176	238	248
B–7	64814	253	46

network A method of linking a number of computers so that they can communicate with each other and share the same peripheral devices. Sometimes known as local area network

(LAN). A network enables different Commodore 64s to use the same printer or disk drive. They are connected by cables to interfaces which usually plug into the **expansion** port. The term also refers to telephone networks which link computers *via* **modems**.

NEW A command which clears a program from memory and resets the variables. NEW is typically used to remove a program from memory before typing in a new one. Generally it is entered as a **direct command** but it could be used at the end of a program, so that the program would erase itself after completing its task. Note that you can not recover a program after NEW has been performed.

NEXT A command used together with **FOR** to indicate the end of a FOR . . . NEXT loop. NEXT can be followed by the variable which acts as the loop counter. If the loop starts with 'FOR S=1 TO 20' the NEXT statement could take the form 'NEXT S'. But the counter variable is optional and only serves to improve legibility. When a program contains several nested loops, adding the counter variable to the end of a NEXT statement helps to show which NEXT is linked to which

FOR.

The following program uses nested loops to read values into an array. The variables A and B in lines 50 and 60 could be omitted:

```
10   DIM AR(2,4)
20   FOR A=0 TO 2
30   FOR B=0 TO 4
40   READ AR(A,B)
50   NEXT B
60   NEXT A
70   DATA 1,3,4,2,4
80   DATA 0,0,9,8,4
90   DATA 6,5,5,7,3
```

A single NEXT can also terminate several nested loops. In this case the variable names must be added in the correct order. The variable attached to the innermost loop should appear first.

```
10   FOR G=1 TO 50
20   FOR H=3 TO 30
30   FOR K=0 TO 100
40   NEXT G,H,K
```

Associated keywords: **FOR; TO; STEP.**

NEXT WITHOUT FOR An **error message**: either a FOR or NEXT is missing, or the program has jumped past a FOR statement

into a loop (possibly because a GOTO statement has mis-directed it).

NMI (non-maskable interrupt.) NMI **interrupts** can not be disabled. On receiving an NMI interrupt the computer jumps to a service routine *via* a vector at 790 and 791.

See **IRQ**.

NOP A 6510 instruction mnemonic with NO oPeration. This instruction does nothing, but as it takes up two **clock** cycles it can be used to adjust timing delays.

Status register	N	V	B	D	I	Z	C
	–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	NOP	EA	1	2

NOT A logical operator which reverses the **truth value** of an expression. It is most commonly used with IF . . . THEN statements. For example, 'IF A>10 THEN PRINT "TOO BIG"' only prints 'TOO BIG' if 'A' is above '10'. 'IF NOT (A>10) THEN PRINT "JUST RIGHT"' prints 'JUST RIGHT' when 'A' is less than or equal to '10'.

NOT also acts as a bitwise operator. It

produces what it is known as the **two's complement** of a number, by reversing each bit and then adding one.

Associated keywords: **AND**; **OR**.

NOT INPUT FILE An **error message**, caused by trying to read a file which has previously been designated for output by an OPEN statement.

NOT OUTPUT An **error message**, caused by trying to write to a file which has previously been designated for input by an OPEN statement.

ON A statement used in conjunction with **GOTO** or **GOSUB** to cause the program to jump to one of a selection line numbers. It has the form ON variable GOTO/GOSUB line number, line number, . . . The value of the variable determines which line number the program jumps to. For example, 'ON X GOTO 100,200,150,2000,1000'. If X equals 3 the program performs a GOTO to line 150, the third item in the list. The alternative to using ON here would be a series of IF . . . THEN statements:

    IF X=1 THEN GOTO 100

```
IF X=2 THEN GOTO 200
IF X=3 THEN GOTO 150
```

The ON construction, however, saves space and takes less time to execute. It can also take an expression involving a variable, as in:

```
ON X+2 GOTO 400,200,50,1000
ON 5*(X=4)-(X<0)*2 GOSUB 100,200,
100,300,500
```

If the value of the variable or expression is zero or greater than the number of line numbers, the program passes on to the next statement. The following program illustrates how ON can be used with a menu giving a choice of mathematical functions.

```
10    PRINT "1. SQUARE ROOT"
20    PRINT "2. SQUARE"
30    PRINT "3. CUBE"
40    PRINT "4. LOG"
50    GET A$:IF A$="" GOTO 50
60    X=VAL(A$)
70    INPUT" TYPE A NUMBER ";N
80  ON X GOSUB 100,200,300,400
90    GOTO 10
100   PRINT "THE SQUARE ROOT OF ";N;
      " IS ";SQR(N)
110   RETURN
200   PRINT "THE SQUARE OF ";N;" IS "; N*N
```

```
210  RETURN
300  PRINT "THE CUBE OF ";N;" IS ";
     N×N×N
310  RETURN
400  PRINT "THE LOG OF ";N;" IS ";LOG(N)
410  RETURN
```
Associated keywords: **GOSUB**; **GOTO**

op code (Operating Code)   The single byte
number that a mnemonic stands for; that part
of a machine code instruction which specifies
the operation to be performed. Op codes are
usually given in **hexadecimal**, e.g. **CMP**'s op
code is D8.

OPEN   An input/output statement. Before
using a printer or creating a data file on tape or
disk, the computer requires you to open a
channel directing the data to or from a specific
device. The command which does this is
OPEN. It is also needed for input/output
operations between the computer and other
devices such as a **modem** or a **plotter**.

NUMBER	DEVICE	DIRECTION
0	keyboard	input
1	cassette	input/output
2	RS232 interface	input/output
3	screen	input/output
4	printer	output

NUMBER	DEVICE	DIRECTION
5	printer	output
6	plotter	output
7	plotter	output
8	disk drive	input/output
9	2nd disk drive	input/output
10–255	not assigned	—

OPEN takes the form 'OPEN file-number, device, command number, "string"'. It is not always necessary to supply all four of these parameters. **Printer commands** generally only contain two of them. The file-number can range from 1 to 255, and simply serves to identify a particular channel. Other input/output commands to the same channel must be followed by the same file-number, e.g., 'OPEN 1,1,2' opens a channel to create a data file on cassette, using 1 as the file-number. The command PRINT#1 will now send data to the cassette.

The device parameter specifies the device being used, e.g. 'OPEN 3,8,15, "SCRATCH:FILE1"' sends a command to disk. 8 is the device number for a disk drive, so it must occur as the second parameter.

DEVICE	DEVICE NUMBER	COMMAND NUMBER	STRING
cassette	1	0 = input	
		1 = output	file name

DEVICE	DEVICE NUMBER	COMMAND NUMBER	STRING
		2 = output with end of tape (EOT)	
modem	2	0	
printer	4 or 5	0 = upper/graphics	control registers
		7 = upper/lower case	
disk	8 to 11	0 = program LOAD	
		1 = program SAVE	drive no: program name
		2–14 = data channel	drive no: file name, file type, read/write
		15 = command channel	command

The third parameter, the command number, indicates what sort of operation is to be performed. In the following example, '7' tells the printer to print in upper/lower case mode rather than in upper case/graphics: 'OPEN 4,4,7'

Lastly, the string parameter has various functions. For cassette files it can be used to give the file a name. With disk drives it can also specify a file type, or contain a command.

See **disk commands**; **sequential files**; **relative files**.

Note that the file number is sometimes called the logical file number, and the OPEN command is said to open a logical file.

Associated keywords: **CMD**; **CLOSE**; **GET#**; **INPUT#**; **PRINT#**.

operand That part of an **assembly lan-**

guage instruction that contains data or the address of data, as opposed to the mnemonic instruction itself, the **operator**, e.g. 'AND $FB'. The operand $FB gives the address in memory of a number rather than the number itself. The term operand is also used to refer to variables, strings or numbers when they are part of an **expression**, e.g. 'IF X > Y THEN . . .' where 'X' and 'Y' are the operands.

operating system The program that supervises all the computer's operations. On the Commodore 64 the operating system is known as the **kernel**.

operator An **assembly language** mnemonic. The term operator is used to distinguish the mnemonic part of an assembly language instruction from the **operand** part. More generally, an operator can be a logical, arithmetic, or relational operator. As such it is part of an **expression** and tells the computer what operation to perform, e.g. +, AND, <.

OR A logical operator which, like AND, can also be used as a bitwise operator. In its capacity as a logical operator it usually appears in IF . . . THEN statements to test two conditions,

e.g.:
```
10 IF A>9 OR B=6 THEN GOSUB 300
```
IF either condition is true or both are true then the program proceeds to the GOSUB statement. OR only returns a value of false if both conditions are false.

As a bitwise operator OR is commonly used to alter one or more bits in a byte by providing a **mask**. Thus 'POKE B,PEEK(B) OR 8' could be used to set bit 3 at location B to 1. OR compares the equivalent bits in two numbers. If either or both are equal to 1 then it gives 1 as a result.

See **truth tables**.

Associated keywords: **AND**; **NOT**

ORA A 6510 instruction mnemonic which performs a logical **OR** between a specified byte and the **accumulator**, leaving the result in the accumulator. It is used as a **mask** to set particular bits to 1, e.g. 'ORA #$0C' sets bits 3 and 2 in the accumulator to 1, and leaves the rest of the byte untouched.

	Status register	N	V	B	D	I	Z	C
		√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	ORA # operand	09	2	2

| | | Status register | N | V | B | D | I | Z | C |
| | | | √ | – | – | – | – | √ | – |

addressing mode	assembly language form	op code	No. bytes	No. cycles
zero page	ORA operand	05	2	3
zero page, X	ORA operand, X	15	2	4
absolute	ORA operand	0D	3	4
absolute, X	ORA operand, X	1D	3	4*
absolute, Y	ORA operand, Y	19	3	4*
(indirect, X)	ORA (operand, X)	01	2	6
(indirect), Y	ORA (operand), Y	11	2	5

* Add 1 on page crossing.

OUT OF DATA   An **error message**: there are not enough data items in a **DATA** statement for the computer to READ.

OUT OF MEMORY   An **error message**: either the program is too big for the available **RAM**, or too many **GOSUBs** have been called but not **RETURNED** from.

OVERFLOW   An **error message**: the result of a calculation is larger than $1.70141884 \times 10^{38}$ – the largest number the computer can handle.

parallel   An **interface** that transmits a number of bits at a time through multiple data lines. The most common form of parallel interface is the Centronics. It has 8 data lines, enabling one character to be sent at a time.

Devices that use this standard require a Centronics interface to be plugged in to one of the Commodore 64's **ports** before they can be connected.

PEEK An integer function which returns the value of a single **byte** at a given **address**. PEEK can be used to examine the contents of any memory location from Ø to 65535, whether in RAM or ROM, e.g., '10 PRINT PEEK(1400)' '10 X=PEEK(53277) AND 16'.

Associated keyword: **POKE**.

peripheral An external device which can be connected to the computer, e.g. **disk drive**, **printer**, **modem**, **joystick**.

PHA A 6510 instruction mnemonic which stores (PusHes) the contents of the **Accumulator** on the top of the **stack**. It is often used to store bytes temporarily; for example, to save the contents of the accumulator before branching to an **interrupt** service routine.

See **PLA**.

Status register	N	V	,	B	D	I	Z	C
	–	–		–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	PHA	48	1	3

PHP A 6510 instruction mnemonic which stores the contents of the **status register** on the top of the **stack**. It is generally used to save the flags before a subroutine call. On return from the subroutine **PLP** restores the status register to its previous condition.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–•–	

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	PHP	08	1	3

pixel The smallest point or dot on the screen which can be controlled by the computer. A computer's screen resolution is measured in terms of the number of pixels it contains. The more pixels there are, the smaller each one is, and the higher the resolution. The Commodore 64 offers a resolution of 320 by 200 pixels.

PLA A 6510 instruction mnemonic which PuLls the first byte off the top of the **stack** and

	Status register	N	V	B	D	I	Z	C
		√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	PLA	68	1	4

loads it into the **Accumulator**. Used to restore the contents of the accumulator and other registers after a subroutine.

plotter A type of **printer** used mainly for graphics output such as graphs and charts. Plotters use a set of pens to transfer ink to paper in different colours. Rather than print a character at a time they draw text and graphics, by moving either the pen or the paper beneath it. The Commodore's printer/plotter, the 1520, can be used both for graphics and program listings. It accepts the same commands as a standard printer – OPEN, PRINT# and CMD – but is assigned device number 6.

PLP A 6510 instruction mnemonic which loads the **status register** with the first byte at the top of the **stack**.

See **PHP**.

Status register		N V B D I Z C from stack			
addressing mode	assembly language form	op code	No. bytes	No. cycles	
implied	PLP	28	1	4	

pointer Pointers are 2-byte locations used by the **operating system** to keep a record of

where a program and its variables are stored. They hold addresses in the order: low byte, high byte, e.g. locations 43 and 44 normally contain the values 1 and 8, which give the address of the first BASIC program line: '1 + 256×8 =2049'.

POINTER ADDRESS	POINTS TO
43,44	start of BASIC program area
45,46	start of BASIC variables
47,48	start of arrays
49,50	end of arrays
51,52	start of strings
53,54	end of strings
55,56	end of BASIC program area

POKE A statement which alters the value of a single byte at a given location in memory. It takes the form 'POKE add,n' where 'add' is an **address** in the range 0 to 65535, and 'n' is a number between 0 and 255. Only addresses in RAM can be POKEd. Attempts to POKE values into ROM will have no effect.

Many of the Commodore 64's features such as **sprites**, **sound** and **high-resolution graphics** are only available by POKEing specific registers in memory. To plot pixels from a BASIC program it is necessary to POKE values into the high-res screen memory. In

character mode it also possible to POKE the screen memory as an alternative to printing to the screen. 'POKE 1024,1:POKE 55296,8' places the letter 'A' at the top–left hand corner of the screen, and colours it orange. '1024' is the first address of the screen memory, '1' is the **screen code** for the letter 'A'. In the second statement '55296' is the first address in the colour memory, while '9' is the colour code for orange.

Associated keyword: **PEEK**.

port Usually a socket or an edge connector, a port provides an entry or exit point for transferring data between the computer and other devices. The Commodore 64 has two **joystick** ports at the side, and, at the back, a **serial port**, an **expansion port**, and a **user port**.

POS An integer function that reports the position of the **cursor** in a line. The value of its **argument** is not important and can be any number. POS is most commonly used for controlling the format of a display. In this example it ensures that characters are not printed beyond column 30. When the cursor reaches column 30 the program prints a car-

riage return character, CHR$(13), which sends
it back to the start of the next line.

```
10   FOR N = 1 TO 200
20   PRINT CHR$(INT(RND(0)×27)+65);
30   IF POS(0) > 30 THEN PRINT CHR$(13)
40   NEXT
```

Associated keywords: **TAB**; **SPC**.

post-indexed indirect addressing

Uses a location in **zero page** as a **vector** to a
base address. It then adds the contents of the Y
register to give the effective address, e.g.
'LDA ($FB),Y'. If locations FB and FC hold
the address of a byte at 0400, and the value of
Y is 7, the effective address is 0407.

pre-indexed indirect addressing

Adds the contents of the X register to an
address in **zero page** which acts as a **vector**.
Unlike **post-indexed addressing** the **index
register** is added to the base address and not to
the address it points to, e.g.

```
LDX #4
STA ($31,X)
```

stores the contents of the accumulator at the
address pointed to by locations $35 and $36.
Note that it takes 2 bytes to store an address in
memory; and they hold the address in reverse

order with the low byte first. If in example 'LDA ($40),Y' locations $40 and $41 contain $55 and $C0 respectively, the address they point to is $C055.

PRINT   A statement that prints characters to the screen. Any characters on the keyboard, whether text or graphics, can be displayed on the screen by enclosing them in quotation marks after PRINT. In addition, the statement prints out the contents of variables, and numbers without quotation marks. PRINT can be followed by a list of different items separated by punctuation marks. These determine the position at which the characters are printed. A semicolon after an item causes the next item to be printed immediately afterwards on the same line. Numbers and numeric variables are, however, followed by a space, while positive numbers are also preceded by a space.

The Commodore 64 treats the screen as if it were divided into four print zones, each 10 characters wide. If the previous item has a comma at the end then the next characters are printed from the start of the next print zone. When the list following PRINT does not contain punctuation then the next PRINT

statement displays characters from the start of a new line. PRINT on its own prints a blank line.

SCREEN DISPLAYS

Character positions: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

PRINT EXAMPLES	SCREEN DISPLAY
10 PRINT "ONE";"TWO"; 20 PRINT "THREE"	ONETWOTHREE
10 PRINT "AWAY";8, "HOME";3	AWAY 8 HOME 3
10 PRINT "FOOD" 20 PRINT 30 PRINT "HEATING"	FOOD HEATING
10 A$="ITEMS" 20 PRINT A$,7,25	ITEMS 7 2 5
10 N = 5:M = 7 20 PRINT N; "X";M;"=";"N*M"	5 X 7 = 35
10 A$ = "FOUR AND" 20 B$ = "TWENTY" 30 C$ = "BLACKBIRDS" 10 PRINT ABC$	FOUR AND TWENTY BLACKBIRDS

PRINT can also take a list of variables without any punctuation between them. It prints them out adjacent to each other as if they were separated by semicolons.

There are some characters which can appear in quotation marks but are not displayed on the screen. These are **control characters** and have a range of different effects on the way the following characters are printed. **Colour control characters**, for example, can be used to change or reverse the colour of output to the screen.

Colour control characters allow one to determine the position at which characters are printed more precisely than by using punctuation marks. Another way of doing this is offered by the TAB, POS, and SPC functions. Preceding the PRINT statement by **CMD** causes output to the screen to be diverted to another device, such as a printer or disk drive.

Associated keywords: **PRINT#**; **TAB**; **POS**; **SPC**

PRINT# An input/output statement which writes data items to an external device, such as a cassette or printer. It is followed by the logical file number given in a previous **OPEN**

statement and a list of **variables** or **strings** or numbers, e.g. 'PRINT#1,"TEST"'

If PRINT# is used to write records to a **sequential file** on tape or disk, each record must be correctly separated, e.g. 'PRINT#1,A$,B$,C$' does not separate the variables 'A$', 'B$', 'C$' but sends them out as one data item with spaces in between. When the file is read back, 'INPUT#1,A$' will read in the contents of all three variables. PRINT# followed by one variable automatically ends the data item with a carriage return character, CHR$(13), which acts as a separator. Commas and semicolons can act as separators if they are enclosed in quotation marks or are assigned to variables themselves, e.g. a list of records can be written in any of the following ways:

```
10   R$=CHR$(13):PRINT#1,A$;R$;B$;R$;C$
10   PRINT#1,A$;",";B$;",";C$
10   R$=CHR$(44):PRINT#1,A$ R$ B$ R$ C$
```

printer A device which transfers text and graphics from the computer to paper. Printers are commonly used to copy programs listings on to paper, or to print out text from a **wordprocessor**. Other applications include taking a copy of the computer's screen display,

known as a screen dump.

Among the various types of printer, dot-matrix printers are the most widely used. They work by controlling a print head which has a matrix of metal points and forms characters out of dots. The print head strikes a ribbon to transfer the character pattern to paper. Thermal printers use the same mechanism but print directly to heat-sensitive paper without a ribbon. Daisy wheel printers work with a font of moulded letters and operate in the same way as typewriters. They give a better quality of printout but are less versatile than dot-matrix printers, which can offer different print sizes and typefaces.

The Commodore's own dot-matrix printers are pre-programmed to print **graphics** characters. They also have the advantage of plugging directly into the **serial port** whereas other makes of printer require an **interface**.

See **plotter; printer commands**.

printer commands OPEN, PRINT#, CMD are used to print text or take listings on any make of printer, e.g. 'OPEN4, 4:PRINT#4,"EXAMPLE"' prints a string; 'OPEN4,4:CMD4:LIST' prints a listing. Note

that after a CMD instruction the computer diverts all output to the printer. A CLOSE statement by itself is not sufficient to close a channel. It is also necessary to send an **empty string** to the printer using 'PRINT#4' without any following characters, e.g. 'CLOSE4:PRINT#4'.

In addition to these commands, there are a number of control codes which generally operate only with Commodore printers, e.g.:
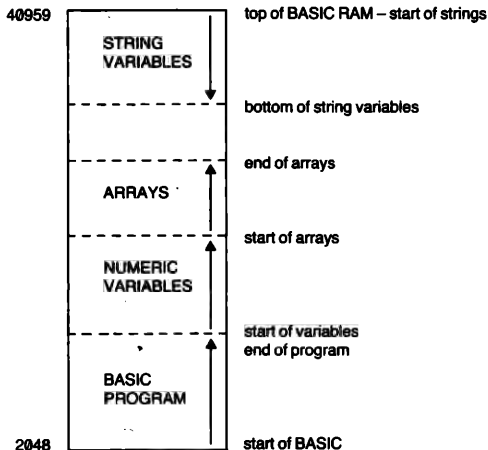PRINT#4,CHR$(14) selects double width characters.
PRINT#4,CHR$(18) prints reverse characters.
PRINT#4,CHR$(17) selects the upper case/lower case character set.

Another way of printing in the alternative **character set** is to give a command number (a secondary address) of 7 as the third parameter in the OPEN statement, e.g. 'OPEN#4,4,7'. To return to upper case/graphics characters use 'OPEN#4,4,0'.

program A sequence of instructions written in a computer language. Programs enable a computer to carry out a task by breaking it down into simple stages.

program area   The area of memory that holds BASIC programs. When the machine is turned on this extends from address 2048 to 40959, giving the user amost 38K. As well as programs, this area also stores their **variables**.

Numeric variables and arrays are stored at the end of a program, while **string variables** are stored from the top of the program area downwards. The **operating system** uses

40959		top of BASIC RAM – start of strings
STRING VARIABLES		
		bottom of string variables
		end of arrays
ARRAYS		
		start of arrays
NUMERIC VARIABLES		
		start of variables
		end of program
BASIC PROGRAM		
2048		start of BASIC

pointers to keep track of the start and end addresses of a program and its variables. By changing the contents of the pointers the program area can be altered; usually in order to reserve memory space for **sprite** or **user-defined character** definitions, or **machine code** programs.

To lower the top of the program area POKE the new address into the locations which point to the top of the program area and the start of string variables – 55,56 and 51,52 – e.g. '10 POKE 56,12:POKE 52,12' lowers the top to 12288. Note that locations 55 and 51 do not need to be altered since they normally contain 0.

To raise the bottom of the program area POKE locations 43 and 44, e.g. 'POKE 4096,0:POKE 44,16:NEW' raises the bottom from 2048 to 4096. This line should be entered as a **direct command**. The first POKE is necessary since the operating system expects to find a 0 at the start of BASIC; NEW resets the other pointers.

program counter A 16-bit **register** which holds the address of the next instruction to be executed. When a **JSR** instruction is

executed the program counter, which holds the address the subroutine will return to, is automatically pushed onto the **stack**. In a branch instruction, such as `BNE` the byte following its **op code** is added to the program counter.

program storage format The way programs are stored in **memory**. The first and last 2 bytes of a program always contain zero. Each program line starts with 2 bytes that hold the address of the next line, known as the link address. Following them the line number is stored in 2 bytes. BASIC keywords are stored as **tokens**; numbers, **strings** and **variables** are stored by their **ASCII** codes. The end of a program line is indicated by a 0. Thus, the program

```
10  PRINT "A"
20  REM
```

is stored in memory as follows:

Address	Contents	Meaning
2048	0	Start of program
2049	11	Link address low byte
2050	8	Link address high byte
2051	10	Line number low byte
2052	0	Line number high byte

Address	Contents	Meaning
2053	153	PRINT token
2054	32	space
2055	34	"
2056	65	A
2057	34	"
2058	0	End of line
2059	17	Link address low byte
2060	8	Link address high byte
2061	20	Line number low byte
2062	0	Line number high byte
2063	143	REM token
2064	0	End of line
2065	0	End of program
2066	0	End of program

Note that the link addresses at 2049 and 2050 point to start of the next line at 2059, i.e. $8 \times 256 + 11 = 2059$.

See **token**; **program area**; **memory map**.

pulse width Specifies the width of a pulse wave peak. It is set by POKEing a value between 0 and 4096 into the high and low pulse width registers. Varying the width gives a different sound quality. A value of 2048 produces a square wave which is often used to synthesise the sound of woodwind instru-

ments such as the clarinet.

RAM (Random Access Memory) The contents of this type of memory can be altered but are not retained when the computer's power supply is switched off. The user's programs and data are stored in RAM. The Commodore 64 has 64K of RAM but only 38K is available for BASIC programs.

See **bank switching**; **memory map**.

raster interrupts **Interrupts** which are triggered by the position of the TV raster beam. The raster beam draws the television image by rapidly scanning each line in turn from top to bottom. As it does so its position is stored in the **VIC** registers at 53265 and 53266. By writing to these registers the user can generate an **IRQ** interrupt every time the beam reaches a specified position. The process also involves setting the interrupt status and interrupt enable registers at 53273 and 53274. Raster interrupts have many applications in **machine code** programs. For example, they can be used to display more than 8 **sprites** at a time.

READ A statement that reads the data given

in a **DATA** statement and assigns it to a
variable. One or a list of variables, separated
by commas, must follow the READ state-
ment. Each must agree with the type of data
expected. Assigning a string data item to a
numeric variable causes a '?SYNTAX
ERROR' message. The computer treats the
DATA statements as a single continuous list,
so it does not matter where a READ statement
is placed or how many statements there are.
Each time a READ is executed it takes the next
item or items in the DATA list. If, for exam-
ple, READ is followed by three variables it
will read the next three items. Attempting to
READ an item when the list has already been
read will cause an 'OUT OF DATA' error
message. READ is commonly used to fill an
array. In the following program it assigns
names to the array A$:

```
10  DIM A$(5)
20  FOR N = 1 TO 5
30  READ A$(N)
40  NEXT
50  DATA JACK,JILL,PETER
60  DATA PAUL,MARY
```

Associated keywords: **DATA**; **RES-
TORE**.

REDIM'D ARRAY An error message, caused by attempting to DIMension an array twice.

REDO FROM START An error message, caused by entering a character string in response to an INPUT prompt when a number is expected. The message will be repeated until the correct input is given.

register Memory locations inside the computer's **microprocessor**. Registers provide temporary storage locations for data and work space for processing data. The 6510 microprocessor has 6 registers: the **accumulator**, 2 **index registers**, the **status register**, the **stack pointer**, and the **program counter**. All the registers are 8 bits wide, with the exception of the program counter which is 16 bits wide. The program counter needs to be twice as wide as the others because its function is to hold the addresses of instructions, With 16 bits (2 bytes), it can hold the address of any location in memory, from 0 to 65535. The sound and video display chips, **SID** and **VIC**, also have their own internal registers, which control sound and screen output. Unlike the 6510's registers these are memory mapped to

RAM. This means that the sound and video chips copy values stored in certain locations in RAM into their internal registers. It allows these registers to be accessed from BASIC.

See **SID**; **VIC**.

bit 7		bit 0
	A	accumulator
	X	index register
	Y	index register
	S	status register
bit 15	SP	stack pointer
PC high byte	PC low byte	program counter
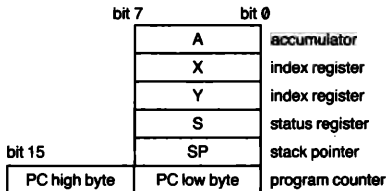
relational operators Used to compare numbers or strings. They usually figure in IF . . . THEN statements, e.g. '10 IF X > Y THEN GOTO 200'.

OPERATOR	MEANING
<	less than
=	equal to
>	greater than
< =	less than or equal to
> =	greater than or equal to
< >	not equal to

They can also be part of an expression which is evaluated as either 'true' or 'false'. If the expression is true, it gives a result of −1; if false, a result of 0 (*see* **truth value**), e.g.

'PRINT 6 > 3' prints '−1'. 'PRINT 5 = 4' prints '∅'. In this capacity, relational operators sometimes provide programming shortcuts, e.g.

```
 50  Y = Y − (Y=4)×10 − (Y=3)×5
```
is equivalent to
```
  50  IF Y = 4 THEN Y=Y+10
  60  IF Y = 3 THEN Y=Y+5
```

relative addressing Used with conditional branch instructions such as **BNE** and **BEQ**. The byte following the **op code** is treated as a displacement (or offset) from the current address, and determines how far forwards or backwards the program branches to. Numbers from ∅ to 127 cause a forward branch; numbers from 128 to 255 cause a branch backwards. The displacement is measured from the end of the instruction, and since the instruction occupies 2 bytes this allow branches in the range +129 to −126 bytes, e.g. 'BNE ∅8' causes a branch to an address 10 bytes forward if the zero flag is ∅. Note that **assemblers** allow labels to be used, making it unnecessary to calculate the displacement, e.g. 'BEQ START'.

relative files A type of data file on **disk**.

Also known as random access files, they allow the user to read or write individual data items (records) without accessing the rest of the file. Relative files are therefore more flexible than **sequential files**, although more difficult to create. Their principal limitation is that each record must be of a fixed length up to 254 bytes. Details of how to create a relative file are given in the 1541 **disk drive** User Guide.

REM A statement used to insert comments or REMarks in a program. Any characters after a REM statement are ignored. Examples are

```
100   REM START OF SOUND SUBROUTINE
100   REM SPRITE DATA
```

Note that colons are treated as part of a REM line, so the statement cannot be placed at the start of a multi-statement line. Thus,

```
10   GOSUB 300:REM BRANCH TO SUBROUTINE
```

causes the program to branch to line 300, but

```
10   REM BRANCH TO SUBROUTINE:
     GOSUB 300
```

does not.

RESTORE A statement used in conjunction with **READ** and **DATA** statements, it instructs the computer to start again at the first

DATA statement. This means that a set of DATA items can be read more than once. The following program reads and prints the same set of data continuously until the **RUN/ STOP key** is pressed:

```
10  RESTORE
20  FOR N = 1 TO 9
30  READ A$
40  PRINT A$
50  NEXT
60  GOTO 10
70  DATA THIS,IS,AN,EXAMPLE,OF
80  DATA THE,USE,OF,RESTORE
```

Associated keywords: **DATA**; **READ**.

RESTORE key Used with the **RUN/ STOP** key to reset the computer. Pressing RUN/STOP and RESTORE together does not erase a program already in memory, but otherwise has the same effect as turning the computer off then on again. Note that the two keys need to be tapped sharply at the same time.

RETURN A statement which marks the end of a subroutine. When the computer meets a RETURN it jumps back to the point in the program which originally called the sub-

routine – the first statement after a GOSUB. A subroutine may have more than one RE-TURN statement in it, to provide several exit points, as in the following example:

```
300  REM START OF SUBROUTINE
310  X = X + Y
320  IF X > 20 THEN RETURN
330  PRINT TAB(X) A$
340  RETURN
```

Associated keyword: **GOSUB**.

RETURN key Pressing this key causes the computer to carry out a **direct command**, or store a program line in its memory.

RETURN WITHOUT GOSUB An error message: either a corresponding GOSUB is missing, or the program has dropped into a subroutine, e.g. because a GOTO statement has mis-directed it.

reverse characters Characters whose foreground and background colours are re-versed. Any characters on the keyboard can be reversed in this fashion. To display reverse characters press the **CTRL key** plus 9 (RVS ON). To return to normal press CTRL plus 0 (RVS OFF). When the RVS ON and RVS

OFF keys are pressed between quotation marks they produce control characters which have the same effect. These can be used to select reverse characters within a program.

Another way of setting RVS ON and RVS OFF is to use the **ASCII** codes for the control characters, e.g. 'PRINT CHR$(18)' turns on reverse mode. The reverse characters themselves have no ASCII codes. Instead, the screen codes 128–255 give the reversed images of codes 0–127.

RIGHT$ A string function used to extract one or more characters from a string, starting from the right-hand end. It takes the form

RIGHT$(A$,N)

where 'A$' is the source string and 'N' is the length of the string to be extracted. Thus,

RIGHT$("PAUCITY",4)

would give 'CITY'. If 'N' is zero it returns an **empty string**. The following program illustrates the way RIGHT$ works by building up a word letter by letter from the right:

```
10  INPUT "TYPE A WORD";A$
20  FOR N=1 TO LEN(A$)
30  PRINT RIGHT$(A$,N)
40  NEXT
```

Associated keywords: **LEFT$; MID$**.

RND A floating-point function. It generates a random number between 0 and 1. If its **argument** is zero the function returns a different number each time, by consulting the system **clock**. Here it is used to give random numbers between 1 and 100:

```
10  PRINT INT(RND(0)×100)+1
20  GOTO 10
```

When its argument is positive the computer generates random numbers by performing calculations on a given initial value, known as a seed. This means that if the same value is used as a seed, RND will return the same sequence of numbers. The seed can be set by using a negative argument. This program prints the same set of numbers every time it is run. Line 10 sets the seed:

```
10  X=RND(-3)
20  FOR N=1 TO 10
30  PRINT RND(1)
40  NEXT
```

ROL A 6510 instruction mnemonic which ROtates the **accumulator** or a given memory location, together with the carry flag, one bit to the Left. It moves the bit in the carry flag to

bit 0, and places bit 7 in the carry flag. It can be used with **ASL** to multiply a multi-byte number by two.

	Status register	N	V	B	D	I	Z	C
		√	–	–	–	–	√	√

addressing mode	assembly language form	op code	No. bytes	No. cycles
accumulator	ROL A	2A	1	2
zero page	ROL operand	26	2	5
zero page, X	ROL operand, X	36	2	6
absolute	ROL operand	2E	3	6
absolute, X	ROL operand, X	3E	3	7

ROM (Read Only Memory)   The contents of this type of memory cannot be altered. Programs or data in ROM are held there permanently. They can be **PEEK**ed, but not **POKE**d. The Commodore 64 has 20K of ROM which contains the **operating system**, the BASIC **interpreter**, and the **character generator**.

ROR   A 6510 instruction mnemonic which ROtates the **accumulator** or a memory location one bit to the Right through the carry flag. When used with **LSR** it has the effect of dividing a multi-byte number by two.
   See overleaf.

	Status register	N	V	B	D	I	Z	C
		√	—	—	—	—	√	√

addressing mode	assembly language form	op code	No. bytes	No. cycles
accumulator	ROR A	6A	1	2
zero page	ROR operand	66	2	5
zero page,X	ROR operand,X	76	2	6
absolute	ROR operand	6E	3	6
absolute,X	ROR operand,X	7E	3	7

RS232 A standard **serial interface** widely
used for transmitting data between the compu-
ter and **peripherals**. The Commodore 64's
serial **port** does not conform to the RS232
standard so devices such as **modems** and
printers often require an RS232 interface be-
fore they can be connected to the computer.
Depending on the application, RS232 inter-
faces plug into the expansion port, the user
port, or the serial port, and sometimes include
software on **cassette**, **disk**, or **cartridge**.

RTI A 6510 instruction mnemonic which
causes a ReTurn from an **IRQ** Interrupt. After
an interrupt has occurred RTI restores control
to a program. In the process it pulls the first
byte off the **stack** and places it in the **status
register** thus restoring the flags to their condi-
tion before the interrupt.

| | Status register | N | V | B | D | I | Z | C |
| | | | | from Stack | | | | |

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	RTI	40	1	6

RTS A 6510 instruction mnemonic which ReTurns from a Subroutine. Used with **JSR** in the same way as the BASIC instructions **GOSUB** and **RETURN**.

| | Status register | N | V | B | D | I | Z | C |
| | | – | – | – | – | – | – | – |

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	RTS	60	1	6

RUN Instructs the computer to start executing the program in its memory. If the command is followed by a line number it executes the program from that line. Otherwise it starts with the first line. RUN also forces the **CLR** statement to be performed. If you wish to avoid this use the **GOTO** statement. The RUN command only starts BASIC and not machine code programs. It is generally entered as a direct command, but it can also be used within a BASIC program.

Associated keywords: **NEW**; **CLR**.

RUN/STOP key Stops a program while it is running. Pressing this key together with SHIFT loads a program from cassette and runs it automatically.

See **RESTORE key**.

RVS *See* **reverse characters**.

SAVE A command used to store a program on tape or disk. SAVE can take three parameters:

    SAVE program name, device, command

The program name must be inclosed in quotation marks unless a string variable is given, e.g.:

    SAVE "PROG1"
    SAVE "PROG1",8
    SAVE A$

When it is not followed by a parameter, SAVE stores a program on tape without a name. The device number specifies disk or tape – 1 for tape, 8 for disk. If no number is given the computer assumes the program is to be stored on tape. If the program is at a different location in memory from normal, a command number of 1 tells the computer to save it so that it LOADs back at the same location, instead of 2048.

See **program area**.

A command number of 2 causes an end of tape marker to be written after the program. When the computer reads this marker it assumes that it has reached the end of the tape (EOT), and displays a 'FILE NOT OPEN' message. A command number of 3 combines the effects of 1 and 2, e.g.: `SAVE "GAME",1,3` adds an EOT marker, and saves the program from a different memory location. Although LOAD can load in machine code programs, SAVE only stores BASIC programs. It cannot be used directly to save a machine code program.

Associated keyword: **LOAD**.

SBC A 6510 instruction mnemonic which SuBtraCts a given value or the contents of a memory location from the contents of the **accumulator**. If the number being subtracted is greater than the number in the accumulator SBC borrows 1 from the carry flag. The carry flag should therefore be set to 1 by a **SEC** instruction before a subtraction, e.g.:

```
SEC
LDA #45
SBC #32
```

subtracts 32 from 45 and leaves 13 in the accumulator. After an SBC instruction the carry flag is set to 0 if a borrow has occurred.

Status register	N	V	B	D	I	Z	C
	√	√	–	–	–	√	√

addressing mode	assembly language form	op code	No. bytes	No. cycles
immediate	SBC # operand	E9	2	2
zero page	SBC operand	E5	2	3
zero page, X	SBC operand, X	F5	2	4
absolute	SBC operand	ED	3	4
absolute, X	SBC operand, X	FD	3	4*
absolute, Y	SBC operand, Y	F9	3	4*
(indirect, X)	SBC (operand, X)	E1	2	6
(indirect), Y	SBC (operand), Y	F1	2	5*

* Add 1 when page boundary is crossed.

screen codes The codes by which characters are represented in **screen memory**. Screen codes are not the same as **ASCII** codes. The following table shows how the two sets of codes are related. (Note that some codes such as those for reverse characters have no corresponding ASCII code.)

UPPER CASE AND FULL GRAPHICS SET	LOWER AND UPPER CASE	UPPER CASE AND FULL GRAPHICS SET	LOWER AND UPPER CASE
0 @	@	3 C	c
1 A	a	4 D	d
2 B	b	5 E	e

UPPER CASE AND FULL GRAPHICS SET		LOWER AND UPPER CASE	UPPER CASE AND FULL GRAPHICS SET		LOWER AND UPPER CASE
6	F	f	33	!	!
7	G	g	34	"	"
8	H	h	35	#	#
9	I	i	36	$	$
10	J	j	37	%	%
11	K	k	38	&	&
12	L	l	39	'	'
13	M	m	40	(	(
14	N	n	41	)	)
15	O	o	42	✳	✳
16	P	p	43	+	+
17	Q	q	44	'	'
18	R	r	45	−	−
19	S	s	46	.	.
20	T	t	47	/	/
21	U	u	48	0	0
22	V	v	49	1	1
23	W	w	50	2	2
24	X	x	51	3	3
25	Y	y	52	4	4
26	Z	z	53	5	5
27	[	[	54	6	6
28	£	£	55	7	7
29	]	]	56	8	8
30	↑	↑	57	9	9
31	←	←	58	:	:
32	space	space	59	;	;

UPPER CASE AND FULL GRAPHICS SET	LOWER AND UPPER CASE		UPPER CASE AND FULL GRAPHICS SET	LOWER AND UPPER CASE
60	<	<	85	U
61	=	=	86	V
62	>	>	87	W
63	?	?	88	X
64			89	Y
65	A		90	Z
66	B		91	
67	C		92	
68	D		93	
69	E		94	
70	F		95	
71	G		96	space	space
72	H		97	
73	I		98	
74	J		99	
75	K		100	
76	L		101	
77	M		102	
78	N		103	
79	O		104	
80	P		105	
81	Q		106	
82	R		107	
83	S		108	
84	T		109	

UPPER CASE AND FULL GRAPHICS SET	LOWER AND UPPER CASE	UPPER CASE AND FULL GRAPHICS SET	LOWER AND UPPER CASE
110		119	
111		120	
112		121	
113		122	
114		123	
115		124	
116		125	
117		126	
118		127	

128–255 reverse-video of 0–127

screen editor The facility that allows a program to be altered or corrected. The Commodore 64 employs a screen editor as opposed to a line editor. It enables a program line to be edited anywhere on the screen. Line editors, by contrast, first require a line to be pulled down to the bottom of the screen. The screen editor is itself a **machine code** program held in **ROM**. To correct a character in a program line, position the **cursor** over it using the cursor keys, and then type in the correction. Characters can be inserted or deleted with the **INST/DEL** key. Pressing the **RETURN key** enters the corrected line into

memory, no matter where the cursor is positioned on the line. To delete a line, type in its number and press RETURN.

screen memory The area of memory where information about what is on the screen is stored. In character mode it runs from 1024 to 2023, although it can be moved to another area of RAM. (*See* **VIC**.) Each byte in the screen memory holds the screen code for the character displayed at the corresponding position on screen.

POKEing a screen code into screen memory causes a character to appear on screen. Similarly **PEEK**ing a location in screen memory reveals a character's code. 'B = 1024 + R$\times$40 + C' gives the screen memory location of a character at column C in row R. This program gives an example of how the screen memory can be PEEKed and POKEd by copying its own listing to the bottom of the screen. Clear the screen, then enter LIST, followed by RUN.

```
10  FOR N=0 TO 159
20  POKE (1024+600+N),PEEK(1024+80+N)
30  POKE (55296+600+N),3
40  NEXT
```

Without line 30 the second listing would not
be visible. It POKEs colour code 3 into the
corresponding locations in colour memory. In
bit map mode the screen memory holds the
information about a high resolution display.
Also known as the bit map, it occupies 8000
bytes and is usually located at address 12288.

scrolling   Also known as fine or smooth
scrolling, this process shifts the display either
vertically or horizontally one pixel at a time.
The **VIC** registers 53265 and 53270 allow the
display to be scrolled 8 times in this manner,
up to one character space. The following
program scrolls text from left to right:

```
10   FOR N=1 TO 40:PRINT "A";:NEXT
20   FOR X=0 TO 7
30   POKE 53270,(PEEK(53270)AND248)+X
40   NEXT
```

Note that as the display moves right it leaves
a space at the left. If the size of the display is
reduced to 38 columns by 24 rows, new data
can be printed so that it scrolls into view from
the left. Combining this technique with a
machine code routine to shift the entire
screen by one character creates a continuous,
smooth, horizontal scrolling effect from left to

right, or *vice versa*.

  'POKE 53270,PEEK(53270)AND 247'
shrinks the screen to 38 columns, blanking out
the columns at either side. 'POKE
53265,PEEK(53265)AND 247)' gives
24 rows.

SEC A 6510 instruction mnemonic which
SEts the Carry flag in the **status register** to 1.
It should always be used before a subtraction
operation with **SBC**.

Status register	N	V	B	D	I	Z	C
	-	-	-	-	-	-	1

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	SEC	38	1	2

SED A 6510 instruction mnemonic which
puts the microprocessor into the decimal mode
by SEtting the Decimal flag to 1.
  See **binary coded decimal**.

Status register	N	V	B	D	I	Z	C
	-	-	-	1	-	-	-

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	SED	F8	1	2

SEI A 6510 instruction mnemonic which

disables **IRQ** interrupts by SEtting the Interrupt flag to 1.

See **CLI**.

	Status register		N	V	B	D	I	Z	C
			–	–	–	–	1	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	SEI	78	1	2

sequential files Used to store a sequence of data on cassette or disk. Sequential files can contain any number of data items (records) of varying lengths. Records are loaded back into the computer in the order in which they were stored; retrieving a particular record means reading through all the records which precede it. A further drawback is that it is not possible to modify a sequential file except by rewriting the entire file or adding a new record to the end. These limitations aside, sequential files are much easier to create than **relative files**.

Apart from a few differences in the format of the **OPEN** command, sequential files are stored on disk in the same way as on cassette.

See **cassette files**.

To write a sequential file to disk, use 'OPEN fn,dn,sa,"FILENAME,S,W"' where

'fn' is the logical file number, 'dn' the device number, '8', and 'sa' the secondary address. 2 is generally given as the secondary address but, unlike the equivalent cassette parameter, it has no significance and can be any number from 2 to 14. 'S' indicates that the file is sequential, 'W' that the file is being written rather than read, e.g.:

```
10  OPEN3,8,2,"NAMES,S,W"
20  PRINT#3,A$
```

Change the 'W' to 'R' to read a file in, e.g.:

```
10  OPEN3,8,2,"NAMES,S,R"
20  INPUT#3,A$
```

serial port A 6-pin DIN socket primarily used to connect Commodore **disk drives** and printers. By plugging one device into another, several disk drives and a printer can be connected to the serial port simultaneously, a technique known as 'daisy-chaining'.

SGN An integer function which indicates whether a number is positive, negative or zero. It gives a result of 1 if the number is positive, -1 if it is negative, and $\emptyset$ if it is zero, e.g.:

```
10  IF SGN(X) = -1 THEN GOTO 200
20  T = SGN(Y)
```

Associated keyword: **ABS**.

SHIFT key   Used with other keys to display the following characters: the symbol at the top of a key; the graphics character at the left of a key; upper case letters in upper/lower case mode. Also used with **Commodore key, cursor keys, CLR/HOME key, INS/DEL, RUN/STOP**.

SID   6581 Sound Interface Device. Controls the computer's sound output. As well as providing three **channels** capable of producing sound over a range of 8 octaves, the SID chip acts as a sound synthesiser. It is capable of creating a variety of different musical and non-musical sounds. The sound chip's registers are represented in **RAM** at addresses 54272 to 54300. They allow the user to define the sound from each channel in terms of its **frequency, envelope, waveform** and **filter**. It is also possible to link two sounds together in various ways, such as by synchronisation and ring modulation. In addition, registers 54297 and 54298 read the positions of two paddles. These devices have the same function as joysticks but with output values from 0 to 255.

REGISTER	ADDRESS	DESCRIPTION
0	54272	voice 1 frequency low byte
1	54273	voice 1 frequency high byte
2	54274	voice 1 pulse width low byte
3	54275	voice 1 pulse width high byte
4	54276	voice 1 waveform type, gate, ring modulation, synchronisation
5	54277	voice 1 attack/decay rate
6	54278	voice 1 sustain/release rate
7	54279	voice 2 frequency low byte
8	54280	voice 2 frequency high byte
9	54281	voice 2 pulse width low byte
10	54282	voice 2 pulse width high byte
11	54283	voice 2 waveform type, gate, ring modulation, synchronisation
12	54284	voice 2 attack/decay rate
13	54285	voice 2 sustain/release rate
14	54286	voice 3 frequency low byte
15	54287	voice 3 frequency high byte
16	54288	voice 3 pulse width low byte
17	54289	voice 3 pulse width high byte
18	54290	voice 3 waveform type, gate, ring modulation, synchronisation
19	54291	voice 3 attack/decay rate
20	54292	voice 3 sustain/release rate
21	54293	filter cutoff frequency high bits (0–2)
22	54294	filter cutoff frequency low byte
23	54295	filter control for voices, resonance (4–7)
24	54296	filter type (4–7), volume
25	54297	X position of games paddle
26	54298	Y position of games paddle
27	54299	digitised output of voice 3 high frequency
28	54300	digitised output of voice 3 waveform

SIN A floating-point function which calcu-

lates the sine of an angle given in radians. To convert an angle from degrees to radians multiply it by PI/180, as in the following program:

```
10  PI = 3.14159
20  INPUT "TYPE AN ANGLE in DEGREES";A
30  PRINT "THE SINE OF ";A;" IS ";
    SIN(A∗PI/180)
```

Associated keywords: **ATN**; **COS**; **TAN**.

software   Another word for a **program** or a set of programs. Often used to contrast a program with the hardware that runs it.

sound   The Commodore 64 can produce sound through three **channels** over a range of 8 octaves. Sound is normally output through the TV but can be also be sent to a hi-fi system *via* the **audio/video port**. The three channels, or **voices**, can produce sound separately or together, enabling three notes to be played at the same time. Each channel is controlled by POKEing its respective register in the **SID** chip. By defining a voice's **frequency**, **waveform** and **envelope**, it is possible to synthesise a wide variety of different sounds.

Setting bit 0 in one of the waveform control registers to 1 turns a voice on, setting it to 0

Sound registers 1–20 and 24			
S=54272			
VOICE 1	VOICE 2	VOICE 3	CONTROLS
S+0	S+7	S+14	frequency low byte
S+1	S+8	S+15	frequency high byte
S+2	S+9	S+16	pulse waveform low byte
S+3	S+10	S+17	pulse waveform high byte
S+4	S+11	S+18	control register for waveform, gate (on/off), ring modulation, synchronisation
S+5	S+12	S+19	attack/decay
S+6	S+13	S+20	sustain/release
S+24	S+24	S+24	volume

turns it off. This bit is known as the gate bit.
The **volume** of a voice cannot be independently controlled but is set for all three voices.
To produce a sound requires a minimum of 6 steps:

1: Set the volume.
2: Define the envelope for a voice.
3: Set the frequency.
4: Select the waveform and turn sound on.
5: Delay for the duration of a note.
6: Turn sound off.

These steps are demonstrated in the following program which runs through the range of

frequencies:
```
10  SD=54272:REM START OF SID
    REGISTERS
20  FOR N=SD TO SD +24
30  POKE N,0:REM CLEAR REGISTERS
40  NEXT N
50  POKE SD+24,15:REM MAXIMUM
    VOLUME
60  POKE SD+5,9:REM DEFINE ENVELOPE
70  POKE SD+6,0:REM FOR VOICE 1
80  POKE SD+4,33:REM TURN ON AND
    SET WAVEFORM
90  FOR F=256 TO 62000 STEP 128
100 FH=INT(F/256):FL=F-256×FH:REM
    ALTER FREQUENCY
110 POKE SD,FL:POKE SD+1,FH:REM SET
    FREQUENCY
120 FOR D=1 TO 100:NEXT:REM DELAY
130 NEXT
140 POKE SD+4,32:TURN SOUND OFF
```

SPC A function used with PRINT to print a given number of spaces on the screen. It takes an argument from 0 to 255. SPC is a useful alternative to TAB for formatting a display. In this example it centres a title on the screen:
```
10  PRINT SPC(10) "COLLINS"
```

```
20  PRINT SPC(20) "MICRO FACTS GEM"
```
Associated keywords: **PRINT**; **TAB**.

speech synthesiser A device which re-produces the sound of human speech. Most speech synthesisers provide a set of allophones – sound units from which almost any word can be built up. Allophones are combinations of phonemes, the basic units of speech. For example, a single allophone might give a standard combination of vowel/consonant sounds. Speech synthesisers generally use a custom speech chip held in a cartridge which plugs into the **expansion port**. They allow the user to create speech from a BASIC pro-gram by representing allophones in a string. In addition, some synthesisers supply a diction-ary in ROM of pre-programmed words, It is also possible to program the **SID** chip to synthesise speech.

As a refinement, a few synthesisers allow each allophone to be given a high or low intonation. Although recognisable, synthe-sised speech is rarely realistic and at best only resembles the human voice.

sprite Like **user defined characters**, sprites are graphic objects whose shape can be

designed by the user. One of the main differences is that the **VIC** chip takes care of sprite movement. When a sprite is given a new position it is deleted at its old position. Another advantage is that sprites can be moved in any direction a pixel a a time. (*See* **sprite collision; sprite expansion; sprite priority; multicoloured sprites**.) Up to 8 sprites can be displayed at a time. They are controlled by POKEing values into the VIC chip's sprite registers, which are located from addresses 53248 to 53294. When dealing with these registers it is easiest to assign the first address to a variable, V, at the start of a program. Thereafter each register can be referred to by adding its number to the variable. For example, to set the colour of sprite '0', POKE the register at 53287 (53248 + 39) by entering:

```
POKE V+39,C
```

Displaying a sprite on screen involves the following stages:

(1) DEFINE SPRITE. Each sprite occupies an area 24 pixels across by 21 pixels down. The shape of a sprite is defined by the bit patterns in a byte. To set the pattern for one row of 24 pixels requires 3 bytes. As there are 21 rows, in

all it takes 21 × 3 or 63 bytes to define a sprite. If a bit is set to 1 then the corresponding pixel in the sprite is coloured in, otherwise it is left blank. The following diagram illustrates how a sprite shape is defined:

row	DATA			row	DATA		
1	0	24	0	7	0	231	0
2	0	24	0	8	1	102	128
3	0	60	0	9	3	102	192
4	0	60	0	10	7	102	224
5	0	126	0	11	7	102	224
6	0	102	0	12	79	102	242

row	DATA			row	DATA		
13	91	102	218	18	0	36	0
14	122	231	94	19	7	231	224
15	106	255	86	20	7	255	224
16	126	255	126	21	4	60	32
17	65	231	130				

The value of each bit depends on its position. To work out the decimal value of each byte add up the values of its 8 bits. For example, in the diagram above the second byte in the first row equals 24 since $0 + 0 + 0 + 16 + 8 + 0 + 0 + 0 = 24$

The data which defines a sprite is stored in memory with the bytes for the first row occupying the first 3 positions, followed by the bytes for successive rows. Normally programs READ and then POKE the 63 numbers into memory from DATA statements. This program takes the data which defines the sprite in the diagram above and POKEs it into memory from 832 onwards:

```
10   REM READ DATA
20   FOR N=0 TO 62
30   READ D:POKE 832+N,D
40   NEXT
50   DATA 0,24,0,0,24,0,0,60,0,0,60,0,0,126,0
60   DATA 0,102,0,0,231,0,1,102,128
70   DATA 3,102,192,7,102,224,7,102,224
```

```
  80  DATA 79,102,242,91,102,218,122,231,94
  90  DATA 106,255,86,126,255,126
 100  DATA 65,231,130,0,36,0,7,231,224
 110  DATA 7,255,244,4,60,32
```

(2) STORE DATA FOR SPRITE DEFINITION. Locations 832 to 1023 are used as a cassette **buffer**. If the cassette is not used during a program this is a convenient place to store the data for up to 3 sprites. Any other free area of RAM can be used so long as its starting address is a multiple of 64. Another suitable area, which can hold a large number of sprite definitions, is from 12288 onwards. This is part of the BASIC program area so if the program is a long one there is a danger that it might overwrite the sprite data. It is advisable therefore to lower the top of the program area to 12287 by making the first line: 10 POKE 55,255:POKE 56,47:CLR

(3) SET SPRITE POINTER TO START OF DATA. The sprite pointer tells the VIC chip where the data is stored. It takes the start address divided by 64. Thus if the definitions for the first sprite, sprite 0, were started from 832, the pointer would be set to 13 since 832 divided by 64 equals 13. The pointer for sprite 0 is located at 2040 and would be set by the

following instructions: 'POKE 2040,13'

V=53248	X POSITION	X POSITION >255	Y POSITION	SPRITE COLOUR	TURN SPRITE ON	POINTER ADDRESS
Sprite 0	V+0	V+16,1	V+1	V+39	V+21,1	2040
Sprite 1	V+2	V+16,2	V+3	V+40	V+21,2	2041
Sprite 2	V+4	V+16,4	V+5	V+41	V+21,4	2042
Sprite 3	V+6	V+16,8	V+7	V+42	V+21,8	2043
Sprite 4	V+8	V+16,16	V+9	V+43	V+21,16	2044
Sprite 5	V+10	V+16,32	V+11	V+44	V+21,32	2045
Sprite 6	V+12	V+16,64	V+13	V+45	V+21,64	2046
Sprite 7	V+14	V+16,128	V+15	V+46	V+21,128	2047

The table of sprite registers gives the pointers for each of the eight sprites. It can be seen that sprite 3 has its pointer at 2043. If its data was stored from 12480 onwards, 'POKE 2043, 195' would set the pointer. 195 is the result of 12480 divided by 64.
Note that altering the pointer for a particular sprite to point to a different 64 byte block of data gives the sprite a different shape. In fact, each sprite can have up to 256 definitions.
(4) SET SPRITE COLOUR. To set a sprite to a particular colour POKE its colour register with the required colour code. (*See* **colour**.) For instance, as 7 is the code for yellow, 'POKE V+41,7' sets sprite 2 to yellow.
(5) TURN ON SPRITE. To turn a sprite on or off

set its corresponding bit in register V+21 to
1 or 0. Thus sprite 3 is controlled by bit 3. As
bit 3 in a byte has a value of 8 to turn on sprite
3:

```
POKE V +21,8
```

Consult the sprite register table above to
find the values which turn on each sprite. Or
use a bit **mask**, as used in the formula:

```
POKE V+21, PEEK(V+21) OR (2 ↑ SN)
```

where SN is the sprite number.

More than one sprite can be turned on at a
time by adding the respective bit values. Bits 3
and 7, for example, have values 16 and 128. So

```
POKE V+21,16+128
```

turns on sprites 16 and 128.

As 255 is represented in binary by 11111111

```
POKE V+21,255
```

turns on all 8 sprites.

Setting a bit in register V+21 to 0 turns the
corresponding sprite off, and can be done by
using this formula

```
POKE V+21,PEEK(V+21) AND
(255−2 ↑ SN)
```

where SN is the sprite number.

(6) SET SPRITE POSITION. The position of a
sprite is controlled by the registers from 53248
to 53264 (V to V+16). (*See* sprite register

table.) By poking these locations a sprite is given a horizontal and vertical position in terms of X and Y coordinates. However sprites are only visible if their X coordinates are within the range 24 to 343, and the Y coordinates are from 50 to 249. Outside these ranges a sprite is off the screen. Positioning a sprite horizontally involves two registers, an X register for each sprite number, and the most significant bit (MSB) register at V+16. Normally the bits at V+16 are set to 0 and the X registers control horizontal positions from 0 to 255. To move a sprite from position 256 to 511 requires that its corresponding bit in V+16 is set to 1. For example:

```
POKE V+16,8:POKE V+6,25
```

puts sprite 3 at X position 300 by setting bit 3 in the MSB X register to 1. Position 300 is calculated by adding 25 to 255.

The following lines can be added to the program above to provide a demonstration:

```
120  V = 53248
125  REM SET POINTER
130  POKE 2040,13,
135  REM SET COLOUR TO RED
140  POKE V+39,2,
145  REM TURN ON SPRITE
```

```
150  POKE V+21,1,
155  REM SET X POSITION
160  POKE V,180,
170  FOR Y = 250 TO 50 STEP −1
175  REM MOVE SPRITE UP
180  POKE V+1,Y,
190  NEXT
200  GOTO 170
```

sprite collision Collisions between sprites or between sprites and other objects are indicated in registers 53278 and 53279, $V+30$ and $V+31$, where $V = 53248$. When two sprites collide their respective bits are set to 1 in $V+30$. To check for a collision use:

IF PEEK (REGISTER) AND X = X THEN . . .

where X is the bit value for a given sprite. For example:

IF PEEK(V+30) AND 2 = 2 THEN . . ..

only takes a specified action if sprite 1 touches another sprite.

V=53248								
sprite–sprite collision: register V+30								
sprite–background collision: register V+31								
bit no./sprite no.	7	6	5	4	3	2	1	0
bit value	128	64	32	16	8	4	2	1

Note that after the register has been read all its bits are reset to 0 again. If it is often a good idea to store the contents of the collision registers in a variable and then test for individual bits. Thus 'CD = PEEK(V+31)' could followed by 'IF CD AND 4 = 4 THEN . . .' to detect whether sprite 3 has collided with a character.

sprite expansion Sprites can be expanded to twice their size, in the horizontal direction, the vertical direction, or both together. Setting the bit, which corresponds to the sprite number, to 1 in register 53277 expands a sprite horizontally.

V=53248								
vertical expansion: register V+23								
horizontal expansion: register V+29								
bit no./sprite no.	7	6	5	4	3	2	1	0
bit value	128	64	32	16	8	4	2	1

For example to expand sprite 5 enter
    POKE V+29,32
where V equals the start of the registers, 53248.

Register 53271 (V + 23) controls vertical expansion, and is set in the same way:

```
POKE V+23,8
```
expands sprite 3 vertically.

Alternatively, use the following formulae, in which SN gives a sprite number between Ø and 7: 'POKE (V+29),PEEK(V+29) OR (2↑SN)' for horizontal expansion. 'POKE (V+23),PEEK(V+23) OR (2↑SN)' for vertical expansion.

To reduce a sprite after expansion use:
```
POKE V+29,PEEK(V+29) AND (255 − 2↑SN)
```
for horizontal reduction, and
```
POKE V+23,PEEK(V+23) AND (255 − 2↑SN)
```
for vertical reduction.

sprite priority   Register 53275 (V+27) determines whether sprites appear to pass behind or in front of other objects on the screen. Setting the bit corresponding to the sprite number to 1 gives any other object on the screen priority over the sprite. This means that the sprite will pass behind other objects. If, for example, the screen shows a program listing in **character mode**. 'POKE V+27,8' causes sprite 3 to appear behind the listing, by setting bit 3 to 1. When the corresponding bit is set to Ø the sprite passes in front of other objects.

Between themselves, lower numbered

sprites have priority over higher numbered sprites. Sprite Ø has the highest priority, sprite 7 the lowest. Thus sprite 4 appears in front of sprite 5.

SQR A floating-point function which returns the square root of a number. It cannot handle negative numbers, e.g.:

```
10  PRINT SQR(81)
10  IF F > SQR(N) THEN GOTO 200
```

STA A 6510 instruction mnemonic which STores the contents of the **Accumulator** at a specified memory location, e.g.:

```
LDY #8
LDA #32
STA $0400,Y
```

stores 32 in location $0408.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
zero page	STA operand	85	2	
zero page, X	STA operand, X	95	2	4
absolute	STA operand	8D	3	4
absolute, X	STA operand, X	9D	3	5
absolute, Y	STA operand, Y	99	3	5
(indirect, X)	STA (operand, X)	81	2	6
(indirect), Y	STA (operand), Y	91	2	6

stack An area of RAM used for temporary storage in machine code programs. The stack extends from addresses $100 to $1FF (256 to 511). It operates on the last in, first out principle, storing numbers on top of each other, and removing them from the top.

Although the first available number is said to be at the top of the stack, it has the lowest address in memory since the stack stores numbers downwards from $1FF. When the 6510 microprocessor places (pushes) a number on the stack, or removes (pulls) a number, the **stack pointer** is automatically decreased or increased to point to the next free space.

See **PHA**; **PHP**; **PLA**; **PLP**.

One of the functions of the stack is to hold the address that a program returns to after a subroutine.

See **JSR**.

stack pointer An 8-bit register which points to the first free location on the 6510 microprocessor's **stack**. When an instruction such as **PHA** pushes a byte onto the stack, the stack pointer is decreased by one. Note that it is decreased rather than increased since the stack expands downwards in memory.

STATUS A function which gives information about input/output operations. It returns a single byte number. Depending on which bits in the byte are set to one, it reports the status of the last operation. (*See* the table of STATUS bit codes below.) STATUS is commonly used in cassette file operations to test for an end of file marker.

STATUS BIT POSITION	STATUS NUMERIC VALUE	CASSETTE READ	SERIAL BUS READ/WRITE	TAPE VERIFY + LOAD
0	1		time out write	
1	2		time out read	
2	4	short block		short block
3	8	long block		long block
4	16	unrecoverable read error		any mismatch
5	32	checksum error		checksum error
6	64	end of file	end of file	
7	−128	end of tape	device not present	end of tape

Associated keywords: **GET#**; **INPUT#**; **PRINT#**.

status register Also known as the processor status register, it holds 7 **flags**. They give various types of information about the

state of the 6510 microprocessor or the effects of the instructions it executes. Each flag corresponds to a bit which can be either 1 or 0. When a bit equals 1 its flag is said to be set; when it equals 0 the flag is clear. From left to right the flags are as follows:
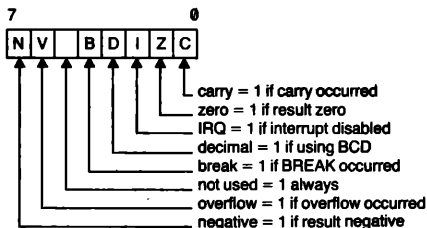
NEGATIVE FLAG (N). Set after an operation when the most significant bit in the result equals 1. In signed arithmetic this indicates that the result is negative. (*See* **two's complement**.)

OVERFLOW FLAG (V). Used in **two's complement** arithmetic to indicate an overflow. It is set when an operation results in a carry from bit 6 to bit 7.

BREAK FLAG (B). Set after a BRK interrupt.

DECIMAL FLAG (D). Set if the 6510 microprocessor is in decimal mode. (*See* **binary coded decimal**)

INTERRUPT FLAG (I). Set to disable an **IRQ** interrupt.

ZERO FLAG (Z). Set when the result of an operation is 0.

CARRY FLAG (C). Set when adding two bytes gives a result greater than 255; cleared if subtracting one byte from another does require a bit to be borrowed. Also acts as a 9th bit for the accumulator in shift and rotate operations.

Bit 5 in the status register is not used and is always set.

STEP A statement which, as part of the **FOR . . . NEXT** loop, STEP allows you to specify the amount by which the loop variable is increased. When STEP is omitted the variable is increased by one. In this line

    10 FOR N = 0 TO 30 STEP 5

the loop variable, N, is increased six times in steps of 5.

    10 FOR N = 10 TO 1 STEP −1

counts down from 10 to 1.

    10 FOR N = 0 TO 10 STEP 0.25

increments the variable in steps of a quarter.

STOP A statement which halts a program and displays a message indicating the line

number where it occurs. Thus

  200   **STOP**

would display 'BREAK IN 200'.

  It has the same effect as pressing the **RUN/ STOP key** during the execution of a program. If the STOP statement is not at the end of a program execution can be resumed by entering **CONT** as a **direct command**.

  Associated keywords: **FOR**; **NEXT**; **TO**.

string   Characters between quotation marks. Strings can hold any combination of letters, numbers, symbols, graphics and control characters, to a maximum of 255 characters.

  See **string variables**.

STRING TOO LONG   An **error message**, caused by trying to form a string longer than 255 characters.

string variables   They store string data. The names of string variables must end with a $ character.

  Two or more string variables can be joined together (concatenated) using the plus sign, e.g.:

  10 **A$ = "HAPPY"**

```
20   B$ = "BIRTHDAY"
30   A$ = A$ + " " + B$
```

They can also be used with **relational operators**, in which case they are compared on the basis of their ASCII codes, e.g. 'IF "4" < "A" THEN PRINT "TRUE"' prints 'TRUE' since 4 has a lower code than A.

CHR$ assigns a single character to a variable, and is often used to insert **control characters** in a string, e.g.:

```
10   A$ = CHR$(146) + CHR$(28) + "TEST"
20   PRINT A$
```

prints the word 'TEST' in blue reverse characters.

STR$ A **string** function which converts numbers into their equivalent string characters. Thus

```
STR$(3.06)
```

gives

```
"3.06"
```

If the number is positive STR$, inserts a space **at** the front of the string. So,

```
PRINT LEN(STR$(3.06)
```

gives a length of 5.

Associated keyword: **VAL**.

structured programming A way of

writing programs so that their structure is evident. Structured programming tries to make programs easy to understand and modify. It does this by breaking the program down into a series of modules or **subroutines**, each one of which handles a specific task. The start of the program can then contain a control section which calls the subroutines and clearly exhibits the flow of the program.

Proponents of structured programming strongly object to the **GOTO** statement. Programs that rely heavily on GOTO are difficult to follow and even harder to modify. However, in Commodore BASIC it is not easy to dispense with GOTO entirely, particularly within a subroutine. Writing properly structured programs requires a set of structured programming commands such as IF . . . THEN . . ELSE, REPEAT . . . UNTIL, and DO . . . WHILE. These are sometimes supplied by **BASIC extensions**.

STX A 6510 instruction mnemonic which STores the contents of the X **index register** in a specified memory location. It acts in the same way as **STA** but has fewer **addressing modes**.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
zero page	STX operand	86	2	3
zero page, Y	STX operand, Y	96	2	4
absolute	STX operand	8E	3	4

STY A 6510 instruction mnemonic which STores the contents of the Y **index register** in a specified memory location.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
zero page	STY operand	84	2	3
zero page, X	STY operand, X	94	2	4
absolute	STY operand	8C	3	4

subroutine One or more program lines which may perform a specific task and can be called from different places within the main program. Subroutines are useful if the same task needs to be performed at several different stages in the program. Instead of repeating a group of lines, it saves space and is more convenient to put them in a subroutine. Alternatively, it is often a good idea to put each stage of a program in a subroutine, even if it is

only used once. The program can then include a control section consisting of a series of **GOSUB** statements.

See **structured programming**.

In BASIC, the GOSUB instruction calls a subroutine and **RETURN** marks the end of a subroutine. The equivalent commands in machine code are **JSR** and **RTS**.

subscript The number inside parentheses by which an element in an **array** is identified. If the subscript is too big for the array it causes a 'BAD SUBSCRIPT' error message, e.g.:

```
10  DIM A(10)
20  A(30) = 2.3
```

sustain/release The last two phases of a **sound envelope**. After the **attack/decay** phase the **volume** falls to the sustain level, and a note continues to play at this level until it is turned off. It then dies away at the rate set for the release phase. Sustain and release for **voices** 1, 2, and 3, are controlled by POKEing values into registers 54278, 54285, 54292. The top four bits of each register set the volume level for the sustain phase, as a proportion of the pre-set volume, e.g. a sustain value of 9 gives a sustain volume which 60% of that set

before the **envelope** is defined (9/15 = 60%). The release value is held in the bottom four bits and acts in the same way as the decay value. It determines the time it takes for a note to fall from its sustain volume to zero, after the gate bit has been set to 0.

SYS A statement which causes the computer to jump to the **machine code** program which starts at the address following SYS. Used either as a direct command or within a BASIC program it is the most common way of executing machine code. When it appears in a BASIC program it has the same effect as a GOSUB except that the program jumps to a machine code program rather than a BASIC subroutine. In this line

    10 SYS 49152:GOTO 300

control passes to the GOTO statement after the machine code at address 49152 has been executed. There must, however, be a **RTS** instruction at the end of the machine code if it is to return to BASIC.

Associated keyword: **USR**.

system variables Locations in **RAM** from 0 to 1023, which are used by the **operating system** and BASIC **interpreter**. Many of

tab 232

them can be usefully PEEKed or POKEd, e.g.
'POKE 650,128' makes all the keys auto-
repeat. 'POKE 198,0' clears the keyboard buf-
fer. 'POKE 646,C' sets the colour of the next
character printed.

TAB A function. Together with PRINT, it
specifies the position at which the next charac-
ter will be printed in a line. It moves the cursor
to a given column position. Thus,
    PRINT TAB(12) "TEST"
prints 'TEST' starting at the thirteenth col-
umn. The left-hand column is numbered 0, the
right-hand column is 39. Although more than
one TAB functions can appear in a single
PRINT statement, TAB cannot be used to
print back to the left, e.g.:
    10   PRINT TAB(5) "ONE" TAB(20) "TWO"
works, but
    10   PRINT TAB(20) "ONE" TAB(5) "TWO"
does not.
    Associated keyword: **PRINT**.

TAN A floating-point function which calcu-
lates the tangent of an angle which is given in
radians. ATN, in turn, gives the angle from its
tangent. Examples:
    10   PRINT TAN(0.5666)

    10 X = TAN(Y)
    Associated keywords: **ATN**; **COS**; **SIN**.

tape   *See* **cassette**.

TAX   A 6510 instruction mnemonic which
Transfers the contents of the **Accumulator** to
the X **index register**. Often used after **PLA**
to restore the contents of the Y register.

Status register   N V B D I Z C
                   √ - - - - √ -

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	TAX	AA	1	2

TAY   A 6510 instruction mnemonic which
Transfers the contents of the **Accumulator** to
the Y **index register**. Often used in conjunc-
tion with **PLA**.

Status register   N V B D I Z C
                   √ - - - - √ -

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	TAY	A8	1	2

TIME   A numeric function which, usually
written 'TI, reads the computer's internal
clock. The clock is set to zero when the

computer is turned on and, thereafter, is increased every 1/60th sec.

TIME is useful for timing intervals. In the following program it measures the amount of time taken to press a key:

```
10   PRINT "PRESS THE FOLLOWING KEY"
20   X = INT(RND(0)*27)
30   PRINT CHR$(65+X)
40   TT = TI
50   GET A$:IF A$ = "" THEN 50
60   PRINT "YOU TOOK"; (TI — TT) / 60;
     " SECONDS"
70   GOTO 10
```

Associated keyword: **TIME$**.

TIME$** Like the TIME function, TIME$ reads the computer's internal **clock but returns a string of six characters which give the elapsed time in hours, minutes, and seconds. Unlike TIME, its initial value can be specified:

```
TIME$ = "HHMMSS"
```

sets the clock to HH hours (up to 24), MM minutes, and SS seconds. The following program sets the timer to 8.30 am and prints a message at 9.00 am:

```
10   TIME$ = "083000"
20   IF TIME$ < "090000" THEN 20
```

30 PRINT "PHONE OFFICE AT ONCE"
Associated keyword: **TIME**.

token The code by which a BASIC
keyword is stored in memory. Rather than
being stored as series of **ASCII** codes, BASIC
keywords are represented in **RAM** by 1-byte
tokens, in the range 128 to 255, e.g. PRINT is
represented by 153. Not only does this save
memory space but it also speeds up the rate at
which programs run. To recognise a keyword
the BASIC **interpreter** needs only to consult a
list of the tokens held in ROM from 41118
onwards. When a program is **LIST**ed,
keywords are converted back into characters
on screen.

truth table   A table showing the results of
comparing different combinations of 1 and 0
using **logical operators**.

1 AND 1=1	1 OR 1=1	NOT 1=0
1 AND 0=0	1 OR 0=1	NOT 0=1
0 AND 1=0	0 OR 1=1	
0 AND 0=0	0 OR 0=0	

Translating 1 and 0 into TRUE and FALSE,
these tables give the results of comparing two
conditions in an IF . . . THEN statement.

truth value The number which the computer assigns to an **expression** depending on whether it is true or false. True expressions are given a value of -1, false expressions a value of $\emptyset$, e.g. 'PRINT A = B' prints '$\emptyset$' if 'A' does not equal 'B'. 'PRINT $6 < 5$' prints '-1'.

In IF . . . THEN statements the truth value acts as a kind of **flag** which gives the result of evaluating the expression. It tells the computer either to execute the instruction after THEN or proceed to the next line. Conversely, when single numbers or variables appear in an IF . . . THEN statement, the computer acts as if they were expressions, and treats them as false if they have a value of $\emptyset$, and true if they have any other value, e.g. 'IF X THEN PRINT "TEST"' prints TEST for all values of X except $\emptyset$.

TSX A 651$\emptyset$ instruction mnemonic which Transfers the **Stack pointer** to the X **index register**. This is the only 651$\emptyset$ microprocessor instruction that allows the contents of the stack pointer to be accessed, e.g.:

```
TSX
STX $FB
```

stores the contents of the stack pointer at FB.

Status register	N	V	B	D	I	Z	C
	↓	–	–	–	–	↓	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	TSX	BA	1	2

turtle *See* LOGO.

two's complement A way of representing negative numbers in **machine code** programs. In two's complement (signed) arithmetic the most significant bit (bit 7) of a byte indicates the sign of a number. If bit 7 is 1 the number is negative; otherwise it is positive. The first 7 bits represent the number itself, giving a range from −128 to +127. Numbers from 0 to 127 ($7F) are considered to be positive, and numbers from 128 to 255 are considered negative. To obtain the two's complement form of a negative number, add it to 256. Thus −100 is 156 in signed arithmetic. In **binary**, first find the complement of the number by inverting (flipping) its bits, then add 1. E.g.:

```
                          binary
                 34       00100010
complement of    34       11011101
          add     1              1
                              _____
               −34        11011110
```

Note that the 6510 microprocessor treats signed numbers in the same way as unsigned numbers. Although signed numbers only occupy the first 7 bits, the 8th bit, bit 7, is set to 1 when two numbers add up to more than 127. This has the effect of giving the result the opposite sign if two numbers with the same sign are added together. To show that an overflow has occured from bit 6 to bit 7 the 6510 sets the overflow (V) flag.

Generally, after an operation, bit 7 in the result is copied into the negative (N) flag. In signed arithmetic this shows whether the result is positive or negative.

TXA A 6510 instruction mnemonic which has the opposite effect to **TAX**, and transfers the X **index register** to the accumulator.

See **PHA**.

Status register		N	V	B	D	I	Z	C
		√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	TXA	8A	1	2

TXS A 6510 instruction mnemonic – the only one that allows the value of the stack pointer to be set, it transfers the contents of the

X index register to the stack pointer.
  See **TSX**.

	Status register	N	V	B	D	I	Z	C
		–	–	–	–	–	–	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	TXS	9A	1	2

TYA A 6510 instruction mnemonic which
Transfers the contents of the Y **index register**
to the **Accumulator**.
  See **PHA**.

	Status register	N	V	B	D	I	Z	C
		√	–	–	–	–	√	–

addressing mode	assembly language form	op code	No. bytes	No. cycles
implied	TYA	98	1	2

TYPE MISMATCH An **error message**: a
number has been used where a string is ex-
pected, or *vice versa*.

UNDEF'D FUNCTION An **error mes-
sage**, caused by trying to use a function which
has not been defined by a **DEF FN** statement.

UNDEF'D STATEMENT An **error mes-
sage**: an attempt has been made to GOTO or
GOSUB to a line number that does not exist.

user defined characters Characters which are designed by the user and replace the built-in characters. The standard character set is defined in the **character generator ROM**. By telling the computer to fetch definitions from an area in RAM it is possible to design up to 512 new characters. To do this requires the following steps:

(1) DEFINE A CHARACTER. Each character is defined by the bit patterns in 8 bytes. Thus it takes 512 bytes to define 64 characters. The bits in the first byte represent the first row, the bits in the second byte the second row, and so on.

See **character designer**.

	64 16 4 1	data
	128 32 8 2	
byte 1		127
byte 2		34
byte 3		20
byte 4		8
byte 5		20
byte 6		34
byte 7		127
byte 8		0

(2) RESERVE MEMORY. Since the new character set is to be held in RAM, memory needs to be allocated for it. From 12288 onwards is a convenient area to store definitions, but runs

the risk of being overwritten by a BASIC program.

```
10  POKE 52,48:POKE 56,48
```

reserves memory by lowering the top of the BASIC **program area**. Other areas of memory can also be used.

(3) CHANGE THE ADDRESS OF CHARACTER MEMORY.

```
20  POKE 53272,(PEEK(53272)AND 240)OR12
```

switches the start address of the character definitions from ROM to 12888. As the new character data has not yet been stored in RAM, any characters on the screen will now be unrecognisable: the computer is taking its definitions from random numbers in RAM.

(4) COPY ROM DEFINITIONS INTO RAM. This step is optional if only user defined characters are needed, but if it is not taken none of the normal character set can be used. For example, if the space character (32) is not defined it will not be possible to clear the screen.

To copy the existing character set, enter these lines:

```
30  POKE 56334,PEEK(56334) AND 254
40  POKE 1,PEEK(1) AND 251
50  FOR N=0 TO 511
60  POKE N+12288,PEEK(53248+N)
```

```
 70   NEXT
 80   POKE 1,PEEK(1) OR 4
 90   POKE 56334,PEEK(56334) OR 1
```

Lines 30 and 40 disable **interrupts** and switch the ROM character set to start at 53248. Lines 50 to 70 then copy the first 64 character into RAM starting at 12288 onwards. Lines 80 and 90 switch out the ROM and enable interrupts.

(5)   STORE NEW CHARACTER DEFINITIONS. Where a character is stored depends on which screen code it is given. The 8 bytes defining a character with code C are stored at location 'CM + (C×8)' onwards where CM is the start of the area of memory reserved for definitions. In this program, which can be added to the lines above, the character defined in the diagram replaces the letter T and is assigned the code 20. Line 110 READs the 8 bytes held in DATA statements, and POKEs them into memory from location 12288 + (20×8) onwards.

```
100   FOR N= 0 TO 7
110   READ D:POKE (12288+20×8+N),D
120   NEXT
130   DATA 127,34,20,8,20,34,127,0
```

(6)   DISPLAY USER DEFINED CHARACTER. When using a PRINT statement simply press the key

associated with the character it replaces. Alternatively, POKE its code into the screen memory. Lines 150 and 160 illustrate both methods. Line 140 clears the screen.

```
140   PRINT CHR$(147)
150   PRINT "T"
160   POKE 1024,20:POKE 55296,6:REM SET
      COLOUR
```

user port The edge connector next to the cassette socket. It has 8 lines for inputting or outputting data and two control lines. Often used to provide an **RS232** or centronics interface, it allows the computer to be connected to a number of different devices, e.g. a **modem**, a **printer**, or a robot arm.

USR A floating-point function which performs in the same way as **SYS** but is less easy to use. It executes a **machine code** program, but before the machine code is called its start address must be placed POKEd into memory locations 785 and 786. Thus,

```
POKE 785,0:POKE 786,192:X = USR(7)
```

calls a machine code routine located at 49152. 785 takes the low order byte of the address and 786 takes the high order byte. In this case 192 is POKEd into 786 since 49152 equals 192

times 256. USR has one advantage over SYS in that it allows a number to be passed from BASIC and used in the machine code program. The number is given as the function's argument. It is placed in the computer's floating point accumulator at locations 97–102. When control returns to BASIC, USR gives the final number stored in the accumulator as a result. In the above example, 7 is passed to the floating-point accumulator and the result is stored in the variable X.

Associated keyword: **SYS**.

utilities Programs that provide useful and commonly needed facilities, often supplied as new commands in BASIC extensions. They assist programmers in the task of writing or modifying a program. The following utilities are among the most common:

RENUMBER renumbers program lines by a given increment.
DELETE deletes a block of program lines.
AUTO prints line numbers automatically.
TRACE a debugging aid which prints the number a line before it is executed.
See **merge**.

VAL A string function which converts a

string which contains a number into the number itself. For example 'VAL("3.55")' gives '3.55'. This function is commonly used to assign numbers held in **string variables** to **numeric variables**. In this program numbers are input to a string variable and then converted to numeric form.

```
10   PRINT "INPUT A NUMBER BETWEEN 0
     AND 10"
20   GET N$:IF N$="" THEN 20
30   IF N$ < "0" OR N$ > "10" THEN 20
40   N = VAL(N$)
```

Note that the first character in the string must be a digit or a plus or minus sign. Otherwise VAL returns zero. Thus 'PRINT VAL(STR$(7.5)' displays '0', as 'STR$(7.5)' inserts a space in front of '7.5'.

   Associated keyword: **STR$**.

variables Used to store data within a program. Each variable is identified by its name which must start with a letter, and can be followed by any number of letters or numbers. There are four kinds of variables – **string, integer, floating point, array** variables. Examples:

   NAME$ – string variable

TT%	–	integer variable
TT	–	floating point variable
N$(4)	–	string array
N%(2)	–	integer array
B1(5)	–	floating point array

Although variable names can be of any length only the first two characters are significant. TEMP$ and TEL$, for example, are treated as the same variable. Long variable names, however, make programs easier to understand.

Variable names must not incorporate BASIC keywords. These are known as reserved words. Using them in a variable will cause a SYNTAX ERROR, e.g. 'TOP = 200' contains the BASIC keyword **TO**.

The equals sign is used to assign a value to a variable, e.g.:

```
A$ = "HELLO"
T2% = 35
N = 3.666
AR(3) = 0.5
```

Variables must take the correct type of value. Attempting to assign a string to a numeric variable – integer and floating point – or *vice versa*, results in a 'TYPE MISMATCH' error message, e.g.:

  B$ = 3
  T = "ALPHA"
  See **VAL**; **STR$**.

vector   A 2-byte location in RAM which
holds the address of another location in mem-
ory. Many of the operating system's sub-
routines in ROM are called indirectly *via* their
vectors in RAM. *See* **JMP**. By changing a
vector to point to a different address the user
can insert a new routine.

  See **wedge**.

VERIFY   (1)   A command used to check
that a program has been correctly **SAVE**d.
VERIFY compares the program stored on tape
or disk with the program in the computer's
memory. If they do not match, it displays a
VERIFY ERROR message. VERIFY on its
own checks the first program on tape.
'VERIFY "PROGNAME"' searches for
'PROGNAME' and checks it if found.
VERIFY '"PROGNAME",8' checks 'PROG-
NAME' on disk.

  This command is also useful for finding the
first unused part of a tape, since it reads the
tape without overwriting the program in
memory.

Associated commands: **SAVE**.

(2)   An **error message**: the program on tape or disk has not been saved correctly, and does not match the program in memory.

VIC   This 6566 Video Interface (VIC II) chip generates the screen display. Whatever the **display mode**, the VIC chip is responsible for converting codes or bits in memory into characters, colours and graphics on the screen. In **character mode** the VIC chip reads character codes in the **screen memory** and then consults the **character generator ROM** to find the pattern of bits which represent characters on screen. Since the computer's microprocessor and the VIC chip cannot access memory at the same time, the 6510's operations are suspended while the VIC generates the display. Although this slows down the 6510, sometimes by as much as 20%, it has no effect on the way the 6510 executes programs. But it can cause problems in **I/O** operations where exact timing is important. It is for this reason that the screen is blanked when the cassette is running.

The VIC chip has 47 registers which are represented in RAM from 53248 to 53294.

Most of them are used for controlling **sprites**, or selecting the display mode. They also provide control over various other features of the display: **screen memory**; screen width and height; fine **scrolling**; screen blanking.

SCREEN MEMORY   The top four bits in VIC register 53272 locate the screen memory at one of sixteen 1K blocks. This allows alternate screens to be set up although it is not possible to shift the location of **colour memory**. Note that the **system variable** at location 648, which points to the screen address, also needs to be changed.

REGISTER	ADDRESS	FUNCTION
0	53248	sprite 0 X-position
1	53249	sprite 0 Y-position
2	53250	sprite 1 X-position
3	53251	sprite 1 Y-position
4	53252	sprite 2 X-position
5	53253	sprite 2 Y-position
6	53254	sprite 3 X-position
7	53255	sprite 3 Y-position
8	53256	sprite 4 X-position
9	53257	sprite 4 Y-position
10	53258	sprite 5 X-position
11	53259	sprite 5 Y-position
12	53260	sprite 6 X-position
13	53261	sprite 6 Y-position
14	53262	sprite 7 X-position
15	53263	sprite 7 Y-position
16	53264	sprites 0–7 most significant bit of X-position

REGISTER	ADDRESS	FUNCTION
17	53265	control register 1
18	53266	raster register
19	53267	light pen X-position
20	53268	light pen Y-position
21	53269	sprites 0–7 enable
22	53270	control register 2
23	53271	sprites 0–7 vertical expansion
24	53272	memory pointers
25	53273	interrupt flag register
26	53274	interrupt enable
27	53275	sprite (0–7)-background priority
28	53276	sprites 0–7 multicolour select
29	53277	sprites 0–7 horizontal expansion
30	53278	sprite (0–7)-sprite collision
31	53279	sprite (0–7)-background collision
32	53280	screen border colour
33	53281	screen background colour
34	53282	background colour 1
35	53283	background colour 2
36	53284	background colour 3
37	53285	sprite multicolour 1
38	53286	sprite multicolour 2
39	53287	sprite 0 colour
40	53288	sprite 1 colour
41	53289	sprite 2 colour
42	53290	sprite 3 colour
43	53291	sprite 4 colour
44	53292	sprite 5 colour
45	53293	sprite 6 colour
46	53294	sprite 7 colour

SCREEN WIDTH AND HEIGHT.   Setting bit 3 to
0 in VIC registers 53265 and 53270 reduces the
screen width to 38 columns and the height to
24 rows, e.g.:

```
POKE 53265,PEEK(53265)AND 247
```

POKE 53270,PEEK(53270)AND 247

FINE SCROLLING.   Controlled by bits 0 to 2 in VIC registers 53265 and 53270.

SCREEN BLANKING.   Setting bit 4 in VIC register 53265 to 0 blanks the screen, e.g. 'POKE 53265,(PEEK(53265)AND239)'. To switch the screen back enter 'POKE 53265, PEEK(53265)OR16'.

See **scrolling**.

REGISTER

NO.	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
17	raster most signif. bit	extended colour mode	bit map mode	screen blanking	screen height	vertical scroll		
22	—	—	—	multi-colour mode	screen width	horizontal scroll		
24	screen memory address				character memory address			

voice   Either a sound **channel** or the sound produced by a channel.

volume   The first four bits of register 54296 control the overall volume of **sound** for the three **channels**. Volume is measured from 0 to 15 where 15 gives a maximum volume and 0 turns the sound off altogether.

WAIT A command which halts a program and waits until a given **address** contains a specified value. It is generally used to test for some external event. For example, it could be used to suspend a program until a key is pressed or a joystick is pushed in a particular direction. Thus 'WAIT 197,28' waits until the B key is pressed. 'WAIT 145,1,1' waits until the joystick in PORT 1 is pushed to the left. It should, however, be noted that there are simpler ways of testing for these events.

WAIT must be followed by an address and one or two numbers which act as **masks**. If the second number is not given it assumes it is zero. WAIT tests the value at the address by comparing it with the first mask in a bitwise AND operation. Then it performs an Exclusive OR operation with the second mask. IF the result of these two operations is 1 the program proceeds to the next statement. In contrast to the normal OR operation, an Exclusive OR gives a result of 1 if only one bit is set to 1. If both bits are 1 the result is 0.

waveform Determines the tonal quality or timbre of a sound. Each **voice** can take one of four waveforms: triangle, sawtooth, pulse,

and noise. The triangle waveform produces a hollow or mellow sound suitable for reproducing a note from a piano or a flute. By contrast the sawtooth sound is more brassy or twangy. Sometimes known as the square wave, the pulse waveform gives a range of different sounds depending on the **pulse width**. The noise waveform is useful for producing non-musical sound effects such as explosions. To assign a waveform to one of the voices, set the appropriate bit in its waveform control register to 1. Note that bit Ø in the same registers turns a sound on or off, e.g. 'POKE 54283,33' selects the sawtooth waveform and turns the sound on.

triangle

sawtooth

pulse

◄—► pulse width

noise
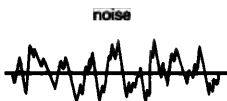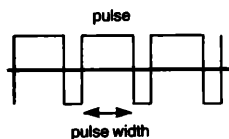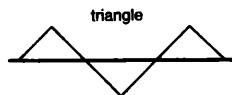
waveform registers 54276, 54283, 54290 – Voices 1, 2, 3		
BIT No.	FUNCTION	BIT VALUE
0	gate (on/off)	1
1	synchronisation	2
2	ring modulation	4
3	test	8
4	triangle	16
5	sawtooth	32
6	pulse	64
7	noise	128

wedge A **machine code** program inserted into one of the **operating system's subroutines**. Wedges are set up by redirecting one of the operating system's **vectors** to point to the user's program. They allow the normal subroutine to be modified or rewritten, e.g. **fast loading** programs insert a wedge in the **input/output** routines. **Interrupt** wedges can be used to harness **IRQ** interrupts so that the user's routine is executed every 1/50th sec. Another type of wedge modifies the BASIC **interpreter's** CHARGET subroutine at locations 115 to 138. CHARGET passes BASIC **tokens** and characters from **RAM** to the Interpreter. By intercepting it, the user can add new

BASIC commands.

wordprocessor A program for entering
text into the computer so that it can be edited,
stored, and printed out. The advantage of a
wordprocessor over a typewriter is that makes
it much easier to correct, rearrange and format
text. All this can be done first on the screen
before a document is printed. As well as
allowing words to be deleted or inserted,
wordprocessors usually provide facilities for
shifting paragraphs, lining up the left or right
margins (justifying text), searching for and
replacing words, taking a word count, and
merging different documents. In some cases
they provide spelling checks from a dictionary
held on disk. Some wordprocessors offer an
80-column option. To run these the
Commodore 64 needs a hardware adaptor
which converts the display to give 80 charac-
ters a line.

zero page Each block of 256 bytes in
memory is known as a page. Zero page is the
block from addresses 0 to 255 (0 to $FF). In
zero page addressing mode instructions move
data to or from (or via) zero page addresses.
The advantage of this mode is that it allows an

address to be specified with one byte rather
than two. Note that when an instruction oper-
ates on a byte in a different page its execution
time is increased by one **clock** cycle, e.g.:

LDX #8
LDA $05FF,X

crosses the boundary between pages 5 and 6,
and so adds one cycle to the normal execution
time.

zero page addressing In this mode the
instruction operates on a byte in **zero page**,
whose address is given by the **operand**. Since
one byte is sufficient to specify any address in
zero page, the whole instruction only occupies
two bytes. By contrast, instructions in **ab-
solute addressing** mode occupy three bytes.
Zero page addressing thus saves space and is
quicker to execute, e.g. 'LDA 56' loads the
aecumulator with the contents of the byte at
location 56; 'AND $FB' performs an 'AND'
operation between the accumulator and the
contents of location FB.

COLLINS GEM

MICRO FACTS

Key Commodore C64 facts simply
accessed and explained
- 6510 instructions
- BASIC keywords ● Sound
- Graphics ● Sprites

£2.25 net

COLLINS
GEM

MICRO
FACTS

COMMODORE 64